HZ BOOKS 华章科技

[PACKT] PUBLISHING

Apache Storm 项目核心贡献者亲笔撰写，涵盖多种分布式计算相关主题，是深入理解Storm分布式实时计算的翔实指南

通过大量的示例，全面而系统地讲解Storm分布式实时计算的核心概念及应用，并针对不同的应用场景，给出多种基于Storm的设计模式，而且提供示例源码，便于读者参考设计并实现自己的Storm应用

大数据 技术丛书

Storm Blueprints
Patterns for Distributed Real-time Computation

Storm分布式实时计算模式

（美） P. Taylor Goetz Brian O'Neill◎著

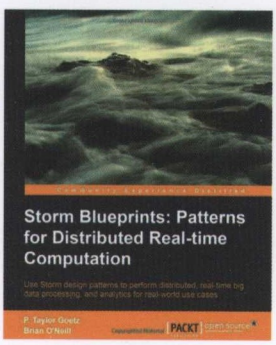
董昭◎译

机械工业出版社
China Machine Press

内容简介

本书由Apache Storm 项目核心贡献者亲笔撰写，融合了作者丰富的Storm实战经验，通过大量示例，全面而系统地讲解使用Storm进行分布式实时计算的核心概念及应用，并针对不同的应用场景，给出多种基于Storm的设计模式，为读者快速掌握Storm分布式实时计算提供系统实践指南。

全书分为10章：第1章介绍使用Storm建立一个分布式流式计算应用所涉及的核心概念，包括Storm的数据结构、开发环境的搭建，以及Storm程序的开发和调试技术等；第2章详细讲解Storm集群环境的安装和搭建，以及如何将topology部署到分布式环境中；第3章通过传感器数据实例详细介绍Trident topology；第4章讲解如何使用Storm和Trident进行实时趋势分析；第5章介绍如何使用Storm进行图形分析，将数据持久化存储在图形数据库中，通过查询数据来发现其中潜在的联系；第6章讲解如何在Storm上使用递归实现一个典型的人工智能算法；第7章演示集成Storm和非事务型系统的复杂性，通过集成Storm和开源探索性分析架构Druid实现一个可配置的实时系统来分析金融事件。第8章探讨Lambda体系结构的实现方法，讲解如何将批处理机制和实时处理引擎结合起来构建一个可纠错的分析系统；第9章讲解如何将Pig脚本转化为topology，并且使用Storm-YARN部署topology，从而将批处理系统转化为实时系统；第10章介绍如何在云服务提供商提供的主机环境下部署和运行Storm。



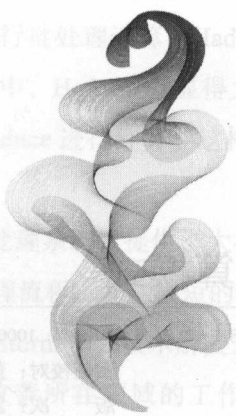
原书封面

技术丛书

Storm Blueprints
Patterns for Distributed Real-time Computation

Storm分布式实时计算模式

（美） P. Taylor Goetz Brian O'Neill◎著
董昭◎译



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Storm 分布式实时计算模式 /（美）吉奥兹（Goetz, P. T.），（美）奥尼尔（O'Neill, B.）著；董昭译 . —北京：机械工业出版社，2014.11（2015.11 重印）
（大数据技术丛书）
书名原文：Storm Blueprints: Patterns for Distributed Real-time Computation

ISBN 978-7-111-48438-7

I. S… II. ①吉… ②奥… ③董… III. 数据处理软件 IV. TP274

中国版本图书馆 CIP 数据核字（2014）第 253835 号

本书版权登记号：图字：01-2014-5893

Storm 分布式实时计算模式

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：吴 怡　　责任校对：董纪丽

印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2015 年 11 月第 1 版第 3 次印刷

开　　本：186mm × 240mm 1/16　　印　　张：16.75

书　　号：ISBN 978-7-111-48438-7　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

The Translator's Words 译者序

大数据概念在各行业已然形成了热潮，犹胜当年的云计算，近期甚至被列入了国家重点发展规划。DataSift 利用 Twitter 上的情感监控预测 Facebook 股价波动，Google 预测世界杯比赛结果，大数据应用的生动案例每每会引发无限遐想：大数据能否对我们所处的行业或领域带来新气象、新思路？迈出尝试的第一步非常重要。

要从海量数据中提取加工对业务有用的信息，选取合适的技术将事半功倍，省去了重新造轮子的烦恼。对海量数据进行批处理运算，Hadoop 依旧保持着无法撼动的地位。但在对实时性要求较高的应用场景中，Hadoop 就显得力不从心。它需要将数据先落地存储到 HDFS 上，然后再通过 MapReduce 进行计算。这样的批处理运算流程使它很难将延时缩小到秒级。

Storm 是基于数据流的实时处理系统，提供了大吞吐量的实时计算能力。每条数据到达系统时，立即在内存中进入处理流程，并在很短的时间内处理完成。实时性要求较高的数据分析场景，都可以尝试使用 Storm 作为技术解决方案。

我们已经决定利用大数据改善所在领域的工作，并选定 Storm 实时流式计算框架作为技术解决方案。这时候的问题是，如何将 Storm 和工作中的实际场景关联起来？这个开源项目的文档并不是非常丰富，源码中示例也很简单。类似的问题可能困扰过不少 Storm 用户。

我在看到本书英文版的介绍时，就感觉到，这正是我想要的，早有这本书能省去多少学习成本！本书并没有非常深入介绍 Storm 的内部实现，而是一本应用指南。其中最有价值的部分，是通过大量翔实的示例，使用 Storm 解决不同的实际应用场景，提出多种基

于 Storm 的设计模式。读者完全可以参考书中示例和源码，来设计并实现自己的 Storm 应用。书中还简要介绍了 Storm 基本概念，以及大规模部署集群的方案，这些都是非常实用的内容。

作为 Storm 的一个忠实用户，能够承担本书的翻译工作实属荣幸。翻译的过程，也是深入学习加深了解的过程。学到作者丰富的 Storm 实践经验，是本次翻译的最大收获。希望这些经验也能够帮助读者少走弯路，快速高效地使用这个工具。

翻译过程中得到了很多人的帮助。首先感谢家人的包容和支持，困难时总有你们的鼓励。感谢腾讯安全平台部的同事们在学习工作中给予的帮助。感谢好友何双宁在翻译过程中提出的建议和探讨。感谢机械工业出版社编辑们的信任和支持。

非常高兴能将这本书分享给大家，也期望有兴趣的朋友一起探讨，共同进步。如果你有任何问题和建议，请联系我 (appledzshr@live.cn)。

Preface 前　言

目前对信息高时效性、可操作性的需求不断增长，这要求软件系统在更少的时间内能处理更多的数据。随着可连接设备数量不断增加，以及在众多行业领域广泛应用，这种信息需求已无处不在。传统企业的运营系统被迫处理原先只有互联网企业才会遇到的大规模数据。这种重大转变正不断瓦解传统架构和解决方案，传统上会将在线事务处理和离线分析分割开来。与此同时，人们正在重新勾勒从数据中提取信息的意义和价值。软件框架和基础设施也在不断进化，以适应这种新场景。

具体地说，数据的生成可以看作一连串发生的离散事件，这些事件流会伴随着不同的数据流、操作和分析，都会由一个通用的软件框架和基础设施来处理。

Storm 正是最流行的实时流计算框架之一，它提供了可容错分布式计算所要求的基本原语和保障机制，可以满足大容量关键业务应用的需求。它不但是一套技术的整合，也是一种数据流和控制的机制。很多大公司都将 Storm 作为大数据处理平台的核心部分。

尝试使用本书中介绍的设计模式，你将学到开发、部署、运营数据处理的流程，它具有每天或者每小时处理上亿次事务的能力。

本书介绍了多种分布式计算相关的主题，包括设计和集成的模式，还介绍了这些技术常见的适用领域和具体应用。本书通过实际示例，从最简单的 topology 出发，首先向用户介绍了 Storm 基础，然后通过更复杂的示例，逐步引入 Storm 的高级概念、更细致的部署方案以及运营中需要关注的事项。

主要内容

第 1 章介绍了使用 Storm 进行分布式流式计算的核心概念。分布式单词计数这个例

子中展示的数据结构、技术和设计模式都是后续进行更复杂计算的基础。在该章中，我们会对 Storm 计算架构有一个基本了解。还将学会搭建 Storm 开发环境，了解开发和调试 Storm 应用的技术。

第 2 章进一步介绍 Storm 技术架构和安装部署 Storm 集群的过程。在该章中，我们会通过配置工具 Puppet 来自动化安装和部署一个多节点 Storm 集群。

第 3 章主要介绍 Trident topology。Trident 在 Storm 基础之上提供了高级抽象，抽象了事务处理和状态管理的细节。该章使用 Trident 框架处理、聚合、过滤来自传感器的数据，以检测传染病是否爆发。

第 4 章介绍使用 Storm 和 Trident 进行实时趋势分析。实时趋势分析引入了在数据流中进行识别的模式。在该章中，你将会整合 Apache Kafka 队列并且通过实现一个滑动窗口来计算移动平均数。

第 5 章介绍了使用 Storm 进行基于图的数据分析，首先将数据持久化到图形数据库，再通过查询数据来发现关系。图形数据库将数据按照顶点、边、属性的图形结构进行存储，聚焦于实体间的关系。在该章中，我们将 Storm 和一种流行的图形数据库 Titan 进行整合，使用 Twitter 作为数据源。

第 6 章介绍在 Storm 上使用递归实现一个典型的人工智能算法。该章展现了 Storm 的局限性，并检视设计模式来适应这些局限。通过分布式远程调用（Distributed Remote Procedure Call，DRPC），你会实现一个提供同步查询服务的 Storm topology，用来决定井字棋游戏下一步怎样走最好。

第 7 章演示整合 Storm 和非事务型系统的复杂性。为了支持这种整合，介绍一种通过 ZooKeeper 进行分布式状态管理的设计模式。该章通过整合 Storm 和开源探索性分析架构 Druid，实现一个可配置的实时系统来分析金融事件。

第 8 章介绍 Lambda 系统架构的概念，结合实时系统和批处理来构建一个可纠错的分析系统。在第 7 章的基础上，你将会融入 Hadoop 的基础设施并且检视如何使用一个 MapReduce job 对 Druid 中可能出现的主机故障事件进行纠错。

第 9 章演示将一个 Hadoop 上运行的 Pig 语言编写的批处理 job 转化为一个实时的 Storm topology。你可以利用 Storm-YARN 来实现这个功能，这个工具可以使用户使用 YARN 来部署和运行 Storm 集群。在 Hadoop 上运行 Storm 系统，企业可以在同一套基础

设施上同时运行与利用实时和批处理系统。

第 10 章提供了在云环境下运行和部署 Storm 系统的最佳实践。详细地说，你可利用一套为云计算服务的库 Apache Whirr，在 Amazon Web Services（AWS）Elastic Compute Cloud（EC2）上部署和配置 Storm 及其相关的支撑组件。此外，你还可以利用 Vagrant 工具在虚拟机环境下建立开发和测试的集群环境。

预备知识

本书中用到的软件如下表所示。

章	节	需要的软件
1		Storm (0.9.1)
2		ZooKeeper (3.3.5) Java (1.7) Puppet (3.4.3) Hiera (1.3.1)
3		Trident (配套 Storm 0.9.1)
4		Kafka (0.7.2) OpenFire (3.9.1)
5		Twitter4J (3.0.3) Titan (0.3.2) Cassandra (1.2.9)
6		无最新软件
7		MySQL (5.6.15) Druid (0.5.58)
8		Hadoop (0.20.2)
9		Storm-YARN (1.0-alpha) Hadoop (2.1.0-beta)
10		Whirr (0.8.2) Vagrant (1.4.3)

面向的读者

初学者和高级用户都可以从本书获益。本书在真实示例的基础上，描述了多种实用的分布式计算模式。书中介绍了 Storm 和 Trident 的核心原语，以及成功部署和运营系统的关键技术。

虽然本书主要讲述 Storm 相关的 Java 开发，但其中的设计模式同样适用于其他编程

语言。书中的小窍门、技术和实现方法对架构师、开发人员和运维人员都具有参考价值。

Hadoop 爱好者会发现，这是一本很好的 Storm 入门书籍，书中举例说明这两种系统如何优势互补，提供了将批处理运算迁移到实时分析的一种高效途径。

本书提供了 Storm 应用于多个问题和行业的具体示例，这些例子应该能够在其他领域中举一反三，解决在有限时间内处理大量数据的问题。同时，解决方案设计师、商业分析师也能从本书介绍的高层系统架构和技术中获益。

读者反馈

我们随时欢迎您的反馈。如果您有任何喜欢或者不喜欢本书的地方，都请告知我们，这将促使我们优化读者最关心的内容。若有反馈，可发送邮件到 feedbak@packtpub.com，邮件标题中请包含本书名。

如果您是某一领域方面的专家，并且有兴趣写一本或者参与一本书，请参考作者指南 www.packtpub.com/authors。

用户支持

对于购买了 Packet 出版社书籍的读者，我们有多种途径帮助你最大限度的利用本书。

下载示例代码

可以在 http://www.packetpub.com 下载到通过 packet 账户购买的所有书籍的示例代码。如果你通过其他途径购买这本书，可以访问 http://www.packetpub.com/support，通过邮箱注册，然后示例代码会通过邮箱发送。

勘误表

尽管我们已经非常用心确保书籍内容的准确性，仍然有错误可能发生。如果您发现书中的任何错误（可能是错字或者代码错误）请将错误报告给我们，不胜感激。这样会避免其他用户对同样的错误造成困惑，有助于我们在后续版本不断完善这本书。勘误报告请在 http://www.packtpub.com/submit-errata 进行上报，先选取本书，点击 errata submission form 链接，然后填入详细的勘误信息即可。勘误内容经确认后就会上传到网站上，或者添加到书籍当前的勘误表中。当前已知的刊物内容可以通过访问 http://www.packtpub.com/support 进行查看。

盗版

互联网上的盗版行为是所有出版媒体共同面临的问题。Packt 出版社采取严厉措施来保护在版权和许可。如果你偶然发现互联网上 Packt 出版社的任何作品，任何形式的非法拷贝，请您立刻提供网址和站点名称以便我们追责。请通过 copyright@packtpub.com 联系我们提供有盗版嫌疑连接。我们非常感谢这种保护作者和出版社权益的帮助，这将有利于提供更有价值的作品。

问题

如果您对本书有任何疑问，可以通过 questions@packtpub.com 联系我们，我们会尽力解答。

作者简介 *About the Author*

P. Taylor Goetz 是 Apache Storm 项目核心贡献者以及发布经理，自 2011 年 10 月 Storm 项目首次开源至今都参与其中，具有长期的 Storm 使用和开发经验。作为 Storm 用户社区中的活跃贡献者，Taylor 领导了一系列开源项目，旨在使企业能够将 Storm 集成到不同的基础设施上。

目前，他在 Hortonworks 公司工作，主导将 Storm 集成到 Hortonworks 的数据平台中 (HDP)。在加入 Hortonworks 公司之前，他就职于 Health Market Science 公司，主导了 Storm 与 HMS 公司的下一代主数据管理平台 (Maste Data Management platform) 的集成工作，涉及 Cassandra 数据库、Kafka 队列、Elastic 搜索引擎以及 Titan 图形数据库等技术。

非常感谢我的妻子、孩子、家人和朋友们，有了你们的爱、支持以及包容和牺牲，才有了这本书。感激不尽。

Brian O'Neill 是一个丈夫、黑客、徒步旅行家、皮划艇爱好者。他还是渔夫、父亲，也是大数据的信徒、开拓者以及分布式计算的梦想家。

他已经担任技术主管超过 15 年，被公认为大数据领域的权威。他作为系统架构师，有着应对各种不同场景的经验，从初创公司到财富五百强公司。他信奉开源精神，对多个项目做出了贡献。他领导的项目，扩展了 Cassandra 数据库，并且将索引引擎，分布式处理框架，分析引擎集成到了该数据库中。他荣获了 2013 年 InfoWorld 的技术领导力奖项。他撰写了关于 Cassandra 的 Dzone 参考文档，被选为 2012 和 2013 年 Datastax Cassandra MVP。

在过去，他作为专家组成员对 Java 社区化进程 (JCP) 做出了贡献，发表过人工智能和上下文发现领域的专利。他对获取了美国布朗大学的计算机科学学士学位引以为荣。

目前，Brian 就职于 Health Market Science(HMS) 公司，任首席技术官，开发的大数据分析平台聚焦于数据管理和医疗领域数据分析。平台主要由 Storm 和 Cassandra 构成，提供了实时数据管理和分析服务。

致我的家人。我的妻子丽莎，我们一起将信仰放飞，它带领我们飞抵云端。孩子们使我们扎根大地，先人们为我们奠定基石，我们的家人成就了我所有的成就。没有你们，这本书永远不可能面世。

审校者简介

Vincent Gijsen 是非常好相处的一个人，他对任何技术相关的事情都非常有耐心。他的背景和感兴趣的领域主要是嵌入式系统工程以及信息科学。他职业生涯初期担任从事市场研究公司的 IT 经理。此后开办了自己的公司，专攻 VOIP 通信技术。现在，他就职于初创公司 ScienceRockstars，该公司聚焦于劝导式用户画像分析和大数据。业余时间他喜欢捣鼓激光发射器、四角直升机模型、eBay 购物、黑客相关的事情，以及啤酒。

Sonal Raj 是一个极客、Python 爱好者、技术狂人。他是 Enfoss 的创办者和行政负责人。他获取了 Jamshedpur 国家技术学院的计算机科学与技术学士学位。曾担任 SERC 的研究员，从事分布式计算和实时运算相关的项目。还在 HCL Infosystems 做过实习生。

他曾在 PyCon India 会议发表过关于 Storm 和 Neo4J 的演讲，在一线杂志和国际期刊上发表过多篇文章和研究论文。

James Xu 是 Apache Storm 项目的核心贡献者，工作在电商领域的 Java/Clojure 工程师。他热衷于新技术比如 Storm 和 Clojure。目前就职于中国电商的领先平台阿里巴巴集团。

目　录 *Contents*

第 1 章 Chapter 1

分布式单词计数

本章将介绍使用 Storm 建立一个分布式流式计算应用时涉及的核心概念。我们通过建立一个简单的计数器程序实现这个目的。计数器将持续输入的一句句话作为输入流，统计其中单词出现的次数。单词计数这个例子浅显易懂，引入了多种数据结构、技术和设计模式。这些都是实现更复杂计算所必须的基础。

本章首先概要介绍 Storm 的数据结构，然后实现一个完整 Storm 程序所需的各个组成部分。读完本章，读者将会了解 Storm 计算的基本结构、搭建开发环境的方法、Storm 程序的开发和调试技术。

本章包括以下主题：

- Storm topology 的基本组成部分——stream、spout 和 bolt。
- 搭建 Storm 开发环境。
- 实现单词计数程序。
- 并发和容错机制。
- 并发计算任务以实现扩容。

1.1 Storm topology 的组成部分——stream、spout 和 bolt

Storm 分布式计算结构称为 topology（拓扑），由 stream（数据流），spout（数据流的生

成者），bolt（运算）组成，如图 1-1 所示。Storm topology 大致等同与 Hadoop 这类批处理运算中的 job。然而，批处理运算中的 job 对运算的起始和终止有着明确定义，Storm topology 会一直运行下去，除非进程被杀死或被取消部署。

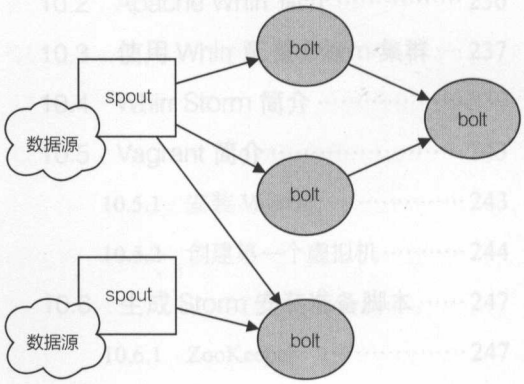


图 1-1

1.1.1 stream

Storm 的核心数据结构是 tuple。tuple 是包含了一个或者多个键值对的列表，Stream 是由无限制的 tuple 组成的序列。如果你对复杂事务处理（Complex Event Processing，CEP）比较熟悉，tuple 就相当于 CEP 中的 event。

1.1.2 spout

spout 代表了一个 Storm topology 的主要数据入口，充当采集器的角色，连接到数据源，将数据转化为一个个 tuple，并将 tuple 作为数据流进行发射。

你会发现 Storm 为实现 spout 提供了非常简单的 API。开发一个 spout 的主要工作就是编写代码从数据源或者 API 消费数据。数据源可能包括以下几种：

- Web 或者移动程序的点击流
- Twitter 或其他社交网络的消息
- 传感器的输出
- 应用程序的日志事件

因为 spout 通常不会用来实现业务逻辑，所以在多个 topology 中常常可以复用。

1.1.3 bolt

bolt 可以理解为计算程序中的运算或者函数，将一个或者多个数据流作为输入，对数据实施运算后，选择性地输出一个或者多个数据流。bolt 可以订阅多个由 spout 或者其他 bolt 发射的数据流，这样就可以建立复杂的数据流转换网络。

像 Spout API 一样，bolt 可以执行各式各样的处理功能，bolt 的编程接口简单明了，

bolt 可以执行的典型功能包括：

- 过滤 tuple
- 连接（join）和聚合操作（aggregation）
- 计算
- 数据库读写

1.2　单词计数 topology 的数据流

如图 1-2 所示，单词计数 topology 由一个 spout 和下游的三个 bolt 组成。

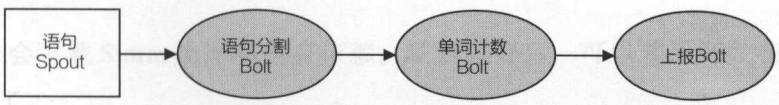


图　1-2

1.2.1　语句生成 spout

SentenceSpout 类的功能很简单，向后端发射一个单值 tuple 组成的数据流，键名是“sentence”，键值是字符串格式存储的一句话。如下所示：

```
{ "sentence":"my dog has fleas" }
```

为了简化起见，我们的数据源是一个静态语句的列表。spout 会一直循环将每句话作为 tuple 发射。实际应用中，spout 通常会连接到动态数据源上，比如通过 Twitter 的 API 获取推特消息。

1.2.2　语句分割 bolt

语句分割 bolt（SplitSentenceBolt）类会订阅 sentence spout 发射的 tuple 流。每当收到一个 tuple，bolt 会获取“sentence”对应值域的语句，然后将语句分割为一个个的单词。每个单词向后发射一个 tuple：

```
{ "word" : "my" }
{ "word" : "dog" }
{ "word" : "has" }
{ "word" : "fleas" }
```

1.2.3 单词计数 bolt

单词计数 bolt（WordCountBolt）订阅 SplitSentenceBolt 类的输出，保存每个特定单词出现的次数。每当 bolt 接收到一个 tuple，会将对应单词的计数加一，并且向后发送该单词当前的计数。

```
{ "word" : "dog", "count" : 5 }
```

1.2.4 上报 bolt

上报 bolt 订阅 WordCountBolt 类的输出，像 WordCountBolt 一样，维护一份所有单词对应的计数的表。当接收到一个 tuple 时，上报 bolt 会更新表中的计数数据，并且将值在终端打印。

1.3 实现单词计数 topology

前面介绍了 Storm 的基础概念，我们已经准备好实现一个简单的应用。现在开始着手开发一个 Storm topology，并且在本地模式执行。Storm 本地模式会在一个 JVM 实例中模拟出一个 Storm 集群。大大简化了用户在开发环境或者 IDE 中进行开发和调试。后续章节将会演示如何将本地模式下开发好的 topology 部署到真实的 Storm 集群环境。

1.3.1 配置开发环境

新建一个 Storm 项目其实就是将 Storm 及其依赖的类库添加到 Java classpath 中。在第 2 章中，你将了解到，将 Storm topology 发布到集群环境中，需要将编译好的类和相关依赖打包在一起。基于这个原因，我们强烈建议使用构建管理工具来管理项目，比如 Apache Maven、Gradle 或者 Leinengen。在单词计数这个例子中，我们使用 Maven。

首先，建立一个 Maven 项目：

```
$ mvn archetype:create -DgroupId=storm.blueprints
-DartifactId=Chapter1 -DpackageName=storm.blueprints.chapter1.v1
```

然后，编辑配置文件 pom.xml，添加 Storm 依赖

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
```

```
    <version>0.9.1-incubating</version>
</dependency>
```

之后，通过执行下述命令编译项目，来测试配置 Maven 是否正确。

```
$ mvn install
```

注意　下载示例代码

如果您是通过 packet 账户购买的书籍，可以在 http://www.packtpub.com 下载到所有已购买书籍的示例代码。如果通过其他途径购买，可以在访问 http://www.packtpub.com/support 并注册。代码会通过邮件发送给您。

Maven 会下载 Storm 类库和所有依赖。项目建好后，可以着手开发我们的第一个 Storm 应用了。

1.3.2　实现 SentenceSpout

为简化起见，SentenceSpout 的实现通过重复静态语句列表来模拟数据源。每句话作为一个单值的 tuple 向后循环发射。完整实现如例 1.1 所示。

例 1.1　SentenceSpout.java

```
public class SentenceSpout extends BaseRichSpout {

    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a cow man",
        "i don't think i like fleas"
    };
    private int index = 0;

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }

    public void open(Map config, TopologyContext context,
                     SpoutOutputCollector collector) {
        this.collector = collector;
    }
```

```
    public void nextTuple() {
        this.collector.emit(new Values(sentences[index]));
        index++;
        if (index >= sentences.length) {
            index = 0;
        }
        Utils.waitForMillis(1);
    }
}
```

BaseRichSpout 类是 ISpout 接口和 IComponent 接口的一个简便的实现。接口对本例中用不到的方法提供了默认实现。使用这个类，我们可以专注在所需要的方法上。方法 declareOutputFields() 是在 IComponent 接口中定义的，所有 Storm 的组件（spout 和 bolt）都必须实现这个接口。Storm 的组件通过这个方法告诉 Storm 该组件会发射哪些数据流，每个数据流的 tuple 中包含哪些字段。本例中，我们声明了 spout 会发射一个数据流，其中的 tuple 包含一个字段（sentence）

Open() 方法在 ISpout 接口中定义，所有 Spout 组件在初始化时调用这个方法。Open() 方法接收三个参数，一个包含了 Storm 配置信息的 map，TopologyContext 对象提供了 topology 中组件的信息，SpoutOutputCollector 对象提供了发射 tuple 的方法。本例中，初始化时不需要做额外操作，因此 open() 方法实现仅仅是简单将 SpoutOutputCollector 对象的引用保存在变量中。

nextTuple() 方法是所有 spout 实现的核心所在，Storm 通过调用这个方法向输出的 collector 发射 tuple。这个例子中，我们发射当前索引对应的语句，并且递增索引指向下一个语句。

1.3.3 实现语句分割 bolt

例 1.2 列出了 SplitSentenceBolt 类的实现。

例 1.2 SplitSentenceBolt.java

```
public class SplitSentenceBolt extends BaseRichBolt{
    private OutputCollector collector;

    public void prepare(Map config, TopologyContext context,
 OutputCollector collector) {
        this.collector = collector;
    }
```

```
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words){
            this.collector.emit(new Values(word));
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

BaseRichBolt 类是 IComponent 和 IBolt 接口的一个简便实现。继承这个类，就不用去实现本例不关心的方法，将注意力放在实现我们需要的功能上。

prepare() 方法在 IBolt 中定义，类同与 ISpout 接口中定义的 open() 方法。这个方法在 bolt 初始化时调用，可以用来准备 bolt 用到的资源，如数据库连接。和 SentenceSpout 类一样，SplitSentenceBolt 类在初始化时没有额外操作，因此 prepare() 方法仅仅保存 OutputCollector 对象的引用。

在 declareOutputFields() 方法中，SplitSentenceBolt 声明了一个输出流，每个 tuple 包含一个字段“word”。

SplitSentenceBolt 类的核心功能在 execute() 方法中实现，这个方法是 IBolt 接口定义的。每当从订阅的数据流中接收一个 tuple，都会调用这个方法。本例中，execute() 方法按照字符串读取“ sentence”字段的值，然后将其拆分为单词，每个单词向后面的输出流发射一个 tuple。

1.3.4　实现单词计数 bolt

WordCountBolt 类（见例 1.3）是 topology 中实际进行单词计数的组件。该 bolt 的 prepare() 方法中，实例化了一个 HashMap<String，Long> 的实例，用来存储单词和对应的计数。大部分实例变量通常是在 prepare() 方法中进行实例化，这个设计模式是由 topology 的部署方式决定的。当 topology 发布时，所有的 bolt 和 spout 组件首先会进行序列化，然后通过网络发送到集群中。如果 spout 或者 bolt 在序列化之前（比如说在构造函数中生成）实例化了任何无法序列化的实例变量，在进行序列化时会抛出 NotSerializableException 异常，topology 就会部署失败。本例中，因为 HashMap<String,Long> 是可序列化的，所以在

构造函数中进行实例化也是安全的。但是，通常情况下最好是在构造函数中对基本数据类型和可序列化的对象进行赋值和实例化，在 prepare() 方法中对不可序列化的对象进行实例化。

在 declareOutputFields() 方法中，类 WordCountBolt 声明了一个输出流，其中的 tuple 包括了单词和对应的计数。execute() 方法中，当接收到一个单词时，首先查找这个单词对应的计数（如果单词没有出现过则计数初始化为 0），递增并存储计数，然后将单词和最新计数作为 tuple 向后发射。将单词计数作为数据流发射，topology 中的其他 bolt 就可以订阅这个数据流进行进一步的处理。

例 1.3 WordCountBolt.java

```
public class WordCountBolt extends BaseRichBolt{
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;

    public void prepare(Map config, TopologyContext context,
            OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }

    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = this.counts.get(word);
        if(count == null){
            count = 0L;
        }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

1.3.5 实现上报 bolt

ReportBolt 类的作用是对所有单词的计数生成一份报告。和 WordCountBolt 类似，ReportBolt 使用一个 HashMap<String,Long> 对象来保存单词和对应计数。本例中，它的功能是简单的存储接收到计数 bolt 发射出的计数 tuple。

上报 bolt 和上述其他 bolt 的一个区别是，它是一个位于数据流末端的 bolt，只接收

tuple。因为它不发射任何数据流，所以 declareOutputFields() 方法是空的。

上报 bolt 中初次引入了 cleanup() 方法，这个方法在 IBolt 接口中定义。Storm 在终止一个 bolt 之前会调用这个方法。本例中我们利用 cleanup() 方法在 topology 关闭时输出最终的计数结果。通常情况下，cleanup() 方法用来释放 bolt 占用的资源，如打开的文件句柄或者数据库连接。

开发 bolt 时需要谨记的是，当 topology 在 Storm 集群上运行时，IBolt.cleanup() 方法是不可靠的，不能保证会执行。下一章讲到 Storm 的容错机制时，会讨论其中的原因。但这个例子我们是运行在开发模式中的，可以保证 cleanup() 被调用。

类 ReportBolt 的完整代码见示例 1.4。

例 1.4　ReportBolt.java

```
public class ReportBolt extends BaseRichBolt {

    private HashMap<String, Long> counts = null;

    public void prepare(Map config, TopologyContext context,
OutputCollector collector) {
        this.counts = new HashMap<String, Long>();
    }

    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = tuple.getLongByField("count");
        this.counts.put(word, count);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // this bolt does not emit anything
    }

    public void cleanup() {
        System.out.println("--- FINAL COUNTS ---");
        List<String> keys = new ArrayList<String>();
        keys.addAll(this.counts.keySet());
        Collections.sort(keys);
        for (String key : keys) {
            System.out.println(key + " : " + this.counts.get(key));
        }
        System.out.println("--------------");
    }
}
```

1.3.6 实现单词计数 topology

我们已经定义了计算所需要的 spout 和 bolt。下面将它们整合为一个可运行的 topology（见例 1.5）

例 1.5 WordCountTopology.java

```
public class WordCountTopology {

    private static final String SENTENCE_SPOUT_ID = "sentence-spout";
    private static final String SPLIT_BOLT_ID = "split-bolt";
    private static final String COUNT_BOLT_ID = "count-bolt";
    private static final String REPORT_BOLT_ID = "report-bolt";
    private static final String TOPOLOGY_NAME = "word-count-topology";

    public static void main(String[] args) throws Exception {

        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();


        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout(SENTENCE_SPOUT_ID, spout);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt(SPLIT_BOLT_ID, splitBolt)
                .shuffleGrouping(SENTENCE_SPOUT_ID);
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt(COUNT_BOLT_ID, countBolt)
                .fieldsGrouping(SPLIT_BOLT_ID, new Fields("word"));
        // WordCountBolt --> ReportBolt
        builder.setBolt(REPORT_BOLT_ID, reportBolt)
                .globalGrouping(COUNT_BOLT_ID);

        Config config = new Config();

        LocalCluster cluster = new LocalCluster();

        cluster.submitTopology(TOPOLOGY_NAME, config, builder.
createTopology());
        waitForSeconds(10);
        cluster.killTopology(TOPOLOGY_NAME);
        cluster.shutdown();
    }
}
```

Storm topology 通常由 Java 的 main() 函数进行定义，运行或者提交（部署到集群的操作）。在本例中，我们首先定义了一系列字符串常量，作为 Storm 组件的唯一标

识符。main() 方法中，首先实例化了 spout 和 bolt，并生成一个 TopologyBuilder 实例。TopologyBuilder 类提供了流式接口风格的 API 来定义 topology 组件之间的数据流。首先注册一个 sentence spout 并且赋值给其唯一的 ID：

```
builder.setSpout(SENTENCE_SPOUT_ID, spout);
```

然后注册一个 SplitSentenceBolt，这个 bolt 订阅 SentenceSpout 发射出来的数据流：

```
builder.setBolt(SPLIT_BOLT_ID, splitBolt)
               .shuffleGrouping(SENTENCE_SPOUT_ID);
```

类 TopologyBuilder 的 setBolt() 方法会注册一个 bolt，并且返回 BoltDeclarer 的实例，可以定义 bolt 的数据源。这个例子中，我们将 SentenceSpout 的唯一 ID 赋值给 shuffleGrouping() 方法确立了这种订阅关系。shuffleGrouping() 方法告诉 Storm，要将类 SentenceSpout 发射的 tuple 随机均匀的分发给 SplitSentenceBolt 的实例。后续在讨论 Storm 的并发性时，会解释数据流分组的详情。代码下一行确立了类 SplitSentenceBolt 和类 theWordCountBolt 之间的连接关系：

```
builder.setBolt(COUNT_BOLT_ID, countBolt)
               .fieldsGrouping(SPLIT_BOLT_ID, new
                               Fields("word"));
```

你将了解到，有时候需要将含有特定数据的 tuple 路由到特殊的 bolt 实例中。在此我们使用类 BoltDeclarer 的 fieldsGrouping() 方法来保证所有“word”字段值相同的 tuple 会被路由到同一个 WordCountBolt 实例中。

定义数据流的最后一步是将 WordCountBolt 实例发射出的 tuple 流路由到类 ReportBolt 上。本例中，我们希望 WordCountBolt 发射的所有 tuple 路由到唯一的 ReportBolt 任务中。globalGrouping() 方法提供了这种用法：

```
builder.setBolt(REPORT_BOLT_ID, reportBolt)
               .globalGrouping(COUNT_BOLT_ID);
```

所有的数据流都已经定义好，运行单词计数计算的最后一步是编译并提交到集群上：

```
Config config = new Config();

LocalCluster cluster = new LocalCluster();

        cluster.submitTopology(TOPOLOGY_NAME, config, builder.
createTopology());
        waitForSeconds(10);
        cluster.killTopology(TOPOLOGY_NAME);
        cluster.shutdown();
```

这里我们采用了 Storm 的本地模式，使用 Storm 的 LocalCluster 类在本地开发环境来模拟一个完整的 Storm 集群。本地模式是开发和测试的简便方式，省去了在分布式集群中反复部署的开销。本地模式还能够很方便地在 IDE 中执行 Storm topology，设置断点，暂停运行，观察变量，分析程序性能。当 topology 发布到分布式集群后，这些事情会很耗时甚至难以做到。

Storm 的 Config 类是一个 HashMap<String, Object> 的子类，并定义了一些 Storm 特有的常量和简便的方法，用来配置 topology 运行时行为。当一个 topology 提交时，Storm 会将默认配置和 Config 实例中的配置合并后作为参数传递给 submitTopology() 方法。合并后的配置被分发给各个 spout 的 bolt 的 open()、prepare() 方法。从这个层面上讲，Config 对象代表了对 topology 所有组件全局生效的配置参数集合。现在可以运行 WordCountTopology 类了，main() 方法会提交 topology，在执行 10 秒后，停止（卸载）该 topology，最后关闭本地模式的集群。程序执行完毕后，在控制台可以看到类似以下的输出：

```
--- FINAL COUNTS ---
a : 1426
ate : 1426
beverages : 1426
cold : 1426
cow : 1426
dog : 2852
don't : 2851
fleas : 2851
has : 1426
have : 1426
homework : 1426
i : 4276
like : 2851
man : 1426
my : 2852
the : 1426
think : 1425
--------------
```

1.4 Storm 的并发机制

在 Storm 的间接中提到过，Storm 计算支持在多台机器上水平扩容，通过将计算切分为多个独立的 tasks 在集群上并发执行来实现。在 Storm 中，一个 task 可以简单地理解为在集群某节点上运行的一个 spout 或者 bolt 实例。

为了理解 storm 的并发机制是如何运行的，我们先来解释下在集群中运行的 topology 的四个主要组成部分：

- Nodes（服务器）：指配置在一个 Storm 集群中的服务器，会执行 topology 的一部分运算。一个 Storm 集群可以包括一个或者多个工作 node。
- Workers（JVM 虚拟机）：指一个 node 上相互独立运行的 JVM 进程。每个 node 可以配置运行一个或者多个 worker。一个 topology 会分配到一个或者多个 worker 上运行。
- Executeor（线程）：指一个 worker 的 jvm 进程中运行的 Java 线程。多个 task 可以指派给同一个 executer 来执行。除非是明确指定，Storm 默认会给每个 executor 分配一个 task。
- Task（bolt/spout 实例）：task 是 spout 和 bolt 的实例，它们的 nextTuple() 和 execute() 方法会被 executors 线程调用执行。

1.4.1　WordCountTopology 的并发机制

到目前为止，在单词计数的示例中没有明确使用任何 Storm 中并发机制的 API，而是让 Storm 使用默认配置。在大多数情况下，除非明确指定，Storm 的默认并发设置默认是 1。

在我们修改 topology 的并发度之前，先来看默认配置下 topology 是如何执行的。假设我们有一台服务器（node），为 topology 分配了一个 worker，并且每个 executer 执行一个 task。我们的 topology 执行过程如图 1-3 所示：

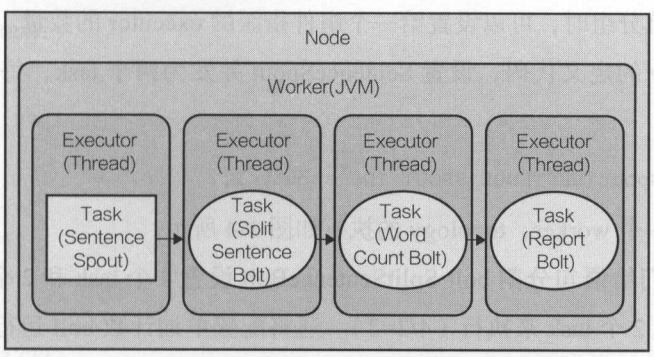


图　1-3

正如在图中看到的，唯一的并发机制出现在线程级。每个任务在同一个 JVM 的不同线程中执行。如何增加并发度以充分利用硬件能力？让我们来增加分配给 topology 的 worker 和 executer 的数量。

1.4.2 给 topology 增加 worker

增加额外的 worker 是增加 topology 计算能力的简单方法。为此 Storm 提供了 API 和修改配置项两种修改方法。无论采取哪种方法，spout 和 bolt 组件都不需要做变更，可以直接复用。

在单词计数 topology 前面的版本中，我们引入了 Config 对象在发布时传递参数给 submitTopology() 方法，但是没有做更多配置操作。为了增加分配给一个 topology 的 worker 数量，只需要简单的调用一下 Config 对象的 setNumWorkers() 方法：

```
Config config = new Config();
config.setNumWorkers(2);
```

这样就给 topology 分配了两个 worker 而不是默认的一个。从而增加了 topology 的计算资源，也更有效的利用了计算资源。我们还可以调整 topology 中的 executor 个数以及每个 executor 分配的 task 数量。

1.4.3 配置 executor 和 task

我们已经知道，Storm 给 topology 中定义的每个组件建立一个 task，默认的情况下，每个 task 分配一个 executor。Storm 的并发机制 API 对此提供了控制方法，允许设定每个 task 对应的 executor 个数和每个 executor 可执行的 task 的个数。

在定义数据流分组时，可以设置给一个组件指派的 executor 的数量。为了说明这个功能，修改 topology 的定义代码，设置 SentenceSpout 并发为两个 task，每个 task 指派各自的 executor 线程。

```
builder.setSpout(SENTENCE_SPOUT_ID, spout, 2);
```

如果只使用一个 worker，topology 的执行如图 1-4 所示。

下一步，我们给语句分割 bolt SplitSentenceBolt 设置 4 个 task 和 2 个 executor。每个 executor 线程指派 2 个 task 来执行（4/2=2）。还将配置单词计数 bolt 运行四个 task，每个 task 由一个 executor 线程执行：

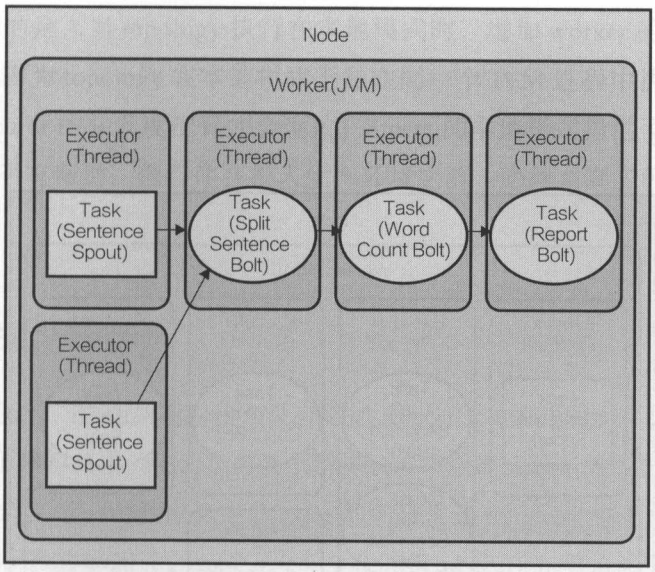


图　1-4

```
builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2)
        .setNumTasks(4)
         .shuffleGrouping(SENTENCE_SPOUT_ID);

builder.setBolt(COUNT_BOLT_ID, countBolt, 4)
         .fieldsGrouping(SPLIT_BOLT_ID, new
          Fields("word"));
```

在 2 个 worker 的情况下，topology 执行如图 1-5 所示。

增加了 topology 并发后，运行更新过的 WordCountTopology 类，每个单词的计数比原 topology 要多：

```
--- FINAL COUNTS ---
a : 2726
ate : 2722
beverages : 2723
cold : 2723
cow : 2726
dog : 5445
don't : 5444
fleas : 5451
has : 2723
have : 2722
homework : 2722
i : 8175
```

```
like : 5449
man : 2722
my : 5445
the : 2727
think : 2722
--------------
```

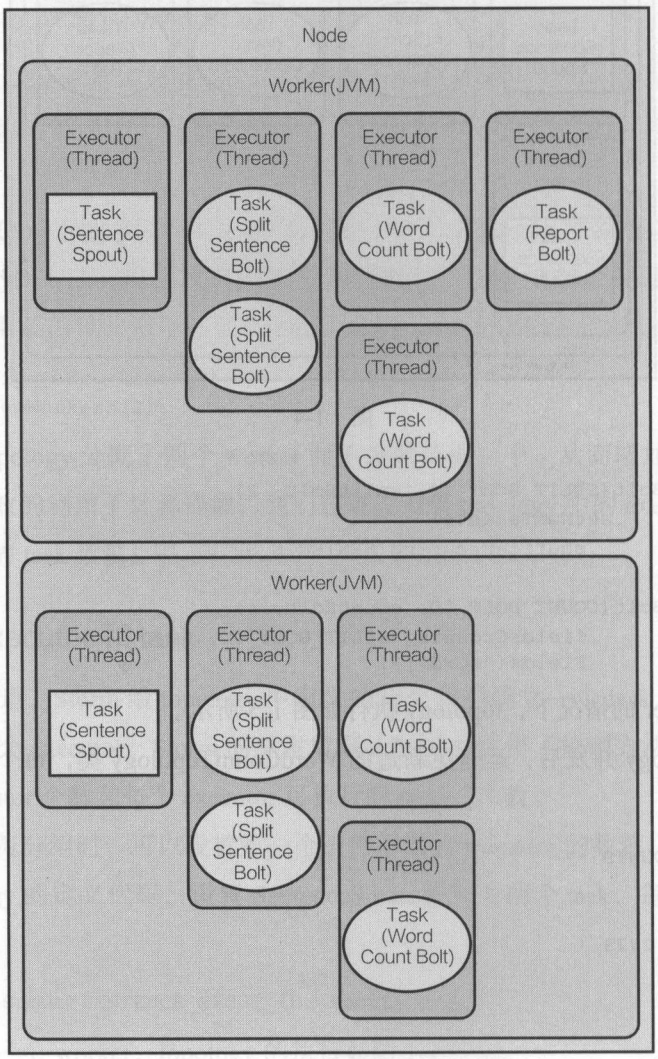


图 1-5

spout 在 topology 关闭之前会一直发射数据，单词的计数值取决于计算机的速度，是否有其他程序在运行。总量上看发射和处理的单词增多了。

要重点指出的是，当 topology 执行在本地模式时，增加 worker 的数量不会达到提高速度的效果。因为 topology 在本地模式下是在同一个 JVM 进程中执行的，所以只有增加 task 和 executor 的并发度配置才会生效。Storm 的本地模式提供了接近集群模式的模拟，对开发是否有帮助。但程序在投入生产环境之前，必须在真实的集群环境下进行测试。

1.5　理解数据流分组

看了前面的例子，你会纳闷为什么没有增加 ReportBolt 的并发度。答案是，这样做没有任何意义。为了理解其中的原因，需要了解 Storm 中数据流分组的概念。

数据流分组定义了一个数据流中的 tuple 如何分发给 topology 中不同 bolt 的 task。举例说明，在并发版本的单词计数 topology 中，SplitSentenceBolt 类指派了四个 task。数据流分组决定了指定的一个 tuple 会分发到哪个 task 上。

Storm 定义了七种内置数据流分组的方式：

- Shuffle grouping（随机分组）：这种方式会随机分发 tuple 给 bolt 的各个 task，每个 bolt 实例接收到的相同数量的 tuple。
- Fields grouping（按字段分组）：根据指定字段的值进行分组。比如说，一个数据流根据“word”字段进行分组，所有具有相同“word”字段值的 tuple 会路由到同一个 bolt 的 task 中。
- All grouping（全复制分组）：将所有的 tuple 复制后分发给所有 bolt task。每个订阅数据流的 task 都会接收到 tuple 的拷贝。
- Globle grouping(全局分组)：这种分组方式将所有的 tuples 路由到唯一一个 task 上。Storm 按照最小的 task ID 来选取接收数据的 task。注意，当使用全局分组方式时，设置 bolt 的 task 并发度是没有意义的，因为所有 tuple 都转发到同一个 task 上了。使用全局分组的时候需要注意，因为所有的 tuple 都转发到一个 JVM 实例上，可能会引起 Storm 集群中某个 JVM 或者服务器出现性能瓶颈或崩溃。
- None grouping（不分组）：在功能上和随机分组相同，是为将来预留的。
- Direct grouping（指向型分组）：数据源会调用 emitDirect() 方法来判断一个 tuple 应该由哪个 Storm 组件来接收。只能在声明了是指向型的数据流上使用。

- Local or shuffle grouping（本地或随机分组）：和随机分组类似，但是，会将 tuple 分发给同一个 worker 内的 bolt task（如果 worker 内有接收数据的 bolt task）。其他情况下，采用随机分组的方式。取决于 topology 的并发度，本地或随机分组可以减少网络传输，从而提高 topology 性能。

除了预定义好的分组方式之外，还可以通过实现 CustomStreamGrouping（自定义分组）接口来自定义分组方式：

```
public interface CustomStreamGrouping extends Serializable {

void prepare(WorkerTopologyContext context,
GlobalStreamId stream, List<Integer> targetTasks);

List<Integer> chooseTasks(int taskId, List<Object> values);
}
```

prepare() 方法在运行时调用，用来初始化分组信息，分组的具体实现会使用这些信息决定如何向接收 task 分发 tuple。WorkerTopologyContext 对象提供了 topology 的上下文信息，GlobalStreamId 提供了待分组数据流的属性。最有用的参数是 targetTasks，是分组所有待选 task 的标识符列表。通常，会将 targetTasks 的引用存在变量里作为 chooseTasks() 的参数。

chooseTasks() 方法返回一个 tuple 发送目标 task 的标识符列表。它的两个参数是发送 tuple 的组件的 id 和 tuple 的值。

为了说明数据流分组的重要性，我们在 topology 中引入一个 bug。首先，修改 SentenceSpout 的 nextTuple() 方法，使每个句子只发送一次：

```
public void nextTuple() {
        if(index < sentences.length){
            this.collector.emit(new Values(sentences[index]));
            index++;
        }
        Utils.waitForMillis(1);
    }
```

程序的输出是这样的：

```
--- FINAL COUNTS ---
a : 2
ate : 2
beverages : 2
cold : 2
cow : 2
dog : 4
don't : 4
fleas : 4
```

```
has : 2
have : 2
homework : 2
i : 6
like : 4
man : 2
my : 4
the : 2
think : 2
--------------
```

然后将 CountBolt 中按字段分组方式修改为随机分组方式：

```
builder.setBolt(COUNT_BOLT_ID, countBolt, 4)
                .shuffleGrouping(SPLIT_BOLT_ID);
```

运行程序的结果是这样的：

```
--- FINAL COUNTS ---
a : 1
ate : 2
beverages : 1
cold : 1
cow : 1
dog : 2
don't : 2
fleas : 1
has : 1
have : 1
homework : 1
i : 3
like : 1
man : 1
my : 1
the : 1
think : 1
--------------
```

结果是错误的，因为 CountBolt 的参数是和状态相关的：它会对收到的每个单词进行计数。这个例子中，在并发状况下，计算的准确度取决于是否按照 tuple 的内容进行适当的分组。我们引入的 bug 只会在 CountBolt 并发实例超过一个时出现。这也是我们为什么一再强调，要在不同的并发度配置下测试 topology。

通常，需要避免将信息存在 bolt 中，因为 bolt 执行异常或者重新指派时，数据会丢失。一种解决方法是定期对存储的信息快照并放在持久性存储中，比如数据库。这样，如果 task 被重新指派就可以恢复数据。

1.6 有保障机制的数据处理

Storm 提供了一种 API 能够保证 spout 发送出来的每个 tuple 都能够执行完整的处理过程。在我们上面的例子中，不担心执行失败的情况。可以看到在一个 topology 中一个 spout 的数据流会被分割生成任意多的数据流，取决于下游 bolt 的行为。如果发生了执行失败会怎样？举个例子，考虑一个负责将数据持久化到数据库的 bolt。怎样处理数据库更新失败的情况？

1.6.1 spout 的可靠性

在 Storm 中，可靠的消息处理机制是从 spout 开始的。一个提供了可靠的处理机制的 spout 需要记录它发射出去的 tuple，当下游 bolt 处理 tuple 或者子 tuple 失败时 spout 能够重新发射。子 tuple 可以理解为 bolt 处理 spout 发射的原始 tuple 后，作为结果发射出去的 tuple。另外一个视角来看，可以将 spout 发射的数据流看作一个 tuple 树的主干（如图 1-6 所示）。

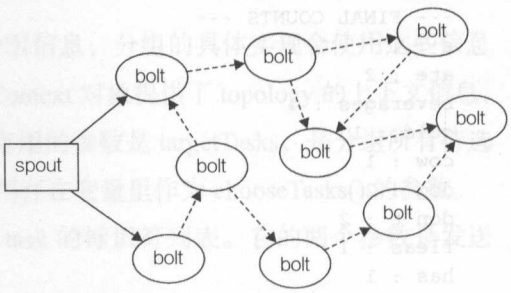


图 1-6

在图中，实线部分表示从 spout 发射的原始主干 tuple，虚线部分表示的子 tuple 都是源自于原始 tuple。这样产生的图形叫做 tuple 树。在有保障数据的处理过程中，bolt 每收到一个 tuple，都需要向上游确认应答（ack）者报错。对主干 tuple 中的一个 tuple，如果 tuple 树上的每个 bolt 进行了确认应答，spout 会调用 ack 方法来标明这条消息已经完全处理了。如果树中任何一个 bolt 处理 tuple 报错，或者处理超时，spout 会调用 fail 方法。

Storm 的 ISpout 接口定义了三个可靠性相关的 API：nextTuple，ack 和 fail。

```
public interface ISpout extends Serializable {
    void open(Map conf, TopologyContext context, SpoutOutputCollector
collector);
    void close();
    void nextTuple();
    void ack(Object msgId);
    void fail(Object msgId);
}
```

前面讲过，Storm 通过调用 Spout 的 nextTuple() 发送一个 tuple。为实现可靠的消息处理，首先要给每个发出的 tuple 带上唯一的 ID，并且将 ID 作为参数传递给 SpoutOutputCollector 的 emit() 方法：

```
collector.emit(new Values("value1", "value2") , msgId);
```

给 tuple 指定 ID 告诉 Storm 系统，无论执行成功还是失败，spout 都要接收 tuple 树上所有节点返回的通知。如果处理成功，spout 的 ack() 方法将会对编号是 ID 的消息应答确认，如果执行失败或者超时，会调用 fail() 方法。

1.6.2　bolt 的可靠性

bolt 要实现可靠的消息处理机制包含两个步骤：

1. 当发射衍生的 tuple 时，需要锚定读入的 tuple

2. 当处理消息成功或者失败时分别确认应答或者报错

锚定一个 tuple 的意思是，建立读入 tuple 和衍生出的 tuple 之间的对应关系，这样下游的 bolt 就可以通过应答确认、报错或超时来加入到 tuple 树结构中。

可以通过调用 OutputCollector 中 emit() 的一个重载函数锚定一个或者一组 tuple：

```
collector.emit(tuple, new Values(word));
```

这里，我们将读入的 tuple 和发射的新 tuple 锚定起来，下游的 bolt 就需要对输出的 tuple 进行确认应答或者报错。另外一个 emit() 方法会发射非锚定的 tuple：

```
collector.emit(new Values(word));));
```

非锚定的 tuple 不会对数据流的可靠性起作用。如果一个非锚定的 tuple 在下游处理失败，原始的根 tuple 不会重新发送。

当处理完成或者发送了新 tuple 之后，可靠数据流中的 bolt 需要应答读入的 tuple：

```
this.collector.ack(tuple);
```

如果处理失败，这样的话 spout 必须发射 tuple，bolt 就要明确地对处理失败的 tuple 报错：

```
this.collector.fail(tuple)
```

如果因为超时的原因，或者显式调用 OutputCollector.fail() 方法，spout 都会重新发送原始 tuple。后面很快有例子。

1.6.3 可靠的单词计数

为了进一步说明可控性，让我们增强 SentenceSpout 类，支持可靠的 tuple 发射方式。需要记录所有发送的 tuple，并且分配一个唯一的 ID。我们使用 HashMap<UUID,Values> 来存储已发送待确认的 tuple。每当发送一个新的 tuple，分配一个唯一的标识符并且存储在我们的 hashmap 中。当收到一个确认消息，从待确认列表中删除该 tuple。如果收到报错，从新发送 tuple：

```
public class SentenceSpout extends BaseRichSpout {

    private ConcurrentHashMap<UUID, Values> pending;
    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a cow man",
        "i don't think i like fleas"
    };
    private int index = 0;

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }

    public void open(Map config, TopologyContext context,
            SpoutOutputCollector collector) {
        this.collector = collector;
        this.pending = new ConcurrentHashMap<UUID, Values>();
    }

    public void nextTuple() {
        Values values = new Values(sentences[index]);
        UUID msgId = UUID.randomUUID();
        this.pending.put(msgId, values);
        this.collector.emit(values, msgId);
        index++;
        if (index >= sentences.length) {
            index = 0;
        }
        Utils.waitForMillis(1);
    }

    public void ack(Object msgId) {
```

```
            this.pending.remove(msgId);
        }

        public void fail(Object msgId) {
            this.collector.emit(this.pending.get(msgId), msgId);
        }
    }
```

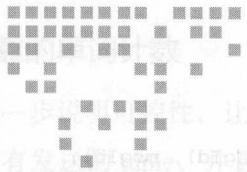
为支持有保障的处理，需要修改 bolt，将输出的 tuple 和输入的 tuple 锚定，并且应答确认输入的 tuple：

```
public class SplitSentenceBolt extends BaseRichBolt{
    private OutputCollector collector;

    public void prepare(Map config, TopologyContext context,
OutputCollector collector) {
        this.collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words){
            this.collector.emit(tuple, new Values(word));
        }
        this.collector.ack(tuple);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

总结

本章中，在没有安装和搭建 Storm 集群的情况下，我们使用 Storm 的核心 API 建立了一个简单的分布式计算程序，覆盖了 Storm 特性集的大部分内容。Storm 的本地模式非常强大，简化了开发，提高了开发效率。但要感受到 Storm 真正的威力和水平扩展性，还是需要将程序部署在真实的集群上。

下一章，我们会讲如何安装和搭建 Storm 集群环境，以及如何将 topology 部署到到分布式环境中。



Chapter 2 第 2 章

配置 Storm 集群

在本章中你将深入理解 Storm 的技术栈，它的软件依赖，以及搭建和部署 Storm 集群的过程。我们首先会在伪分布式模式下安装 Storm，所有的组件都安装在同一台机器上，而不是在多台机器上。一旦你了解了安装和配置 Storm 的基本步骤，我们就可以通过 Puppet 这个工具进行自动化的安装，这样的话部署多节点的集群可以节省大量的时间和精力。

本章包括以下内容：

- 组成 Storm 集群的不同组件和服务
- Storm 的技术栈
- 在 Linux 上安装和配置 Storm
- Storm 的配置参数
- Storm 的命令行接口
- 使用服务提供工具 Puppet 进行自动安装

2.1 Storm 集群的框架

Storm 集群遵循主 / 从（master/slave）结构，和 Hadoop 等分布式计算技术类似，语义上稍有不同。主 / 从结构中，通常有一个配置中静态指定或运行时动态选举出的主节点。

Storm 使用前一种实现方式。主 / 从结构中因为引入了单点故障的风险而被诟病，我们会解释 Storm 的主节点是半容错的。

Storm 集群由一个主节点（称为 nimbus）和一个或者多个工作节点（称为 supervisor）组成。在 nimbus 和 supervisor 节点之外，Storm 还需要一个 Apache ZooKeeper 的实例，ZooKeeper 实例本身可以由一个或者多个节点组成。如图 2-1 所示。

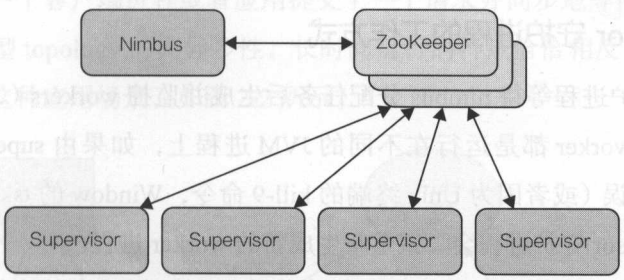


图　2-1

nimbus 和 supervisor 都是 Storm 提供的后台守护进程，可以共存在同一台机器上。实际上，可以建立一个单节点伪集群，把 nimbus、supervisor 和 ZooKeeper 进程都运行在同一台机器上。

2.1.1　理解 nimbus 守护进程

nimbus 守护进程的主要职责是管理，协调和监控在集群上运行的 topology。包括 topology 的发布，任务指派，事件处理失败时重新指派任务。

将 topology 发布到 Storm 集群，将预先打包成 jar 文件的 topology 和配置信息提交（submitting）到 nimbus 服务器上。一旦 nimbus 接收到了 topology 的压缩包，会将 jar 包分发到足够数量的 supervisor 节点上。当 supervisor 节点接收到了 topology 压缩文件，nimbus 就会指派 task（bolt 和 spout 实例）到每个 supervisor 并且发送信号指示 supervisoer 生成足够的 worker 来执行指派的 task。

nimbus 记录所有 supervisor 节点的状态和分配给它们的 task。如果 nimbus 发现某个 supervisor 没有上报心跳或者已经不可达了，它会将故障 supervisor 分配的 task 重新分配到集群中的其他 supervisor 节点。

前面提到过，严格意义上讲 nimbus 不会引起单点故障。这个特性是因为 nimubs 并

不参与 topology 的数据处理过程，它仅仅是管理 topology 的初始化，任务分发和进行监控。实际上，如果 nimbus 守护进程在 topology 运行时停止了，只要分配的 supervisor 和 worker 健康运行，topology 一直继续数据处理。要注意的是，在 nimbus 已经停止的情况下 *supervisor 异常终止，因为没有 nimbus 守护进程来重新指派失败这个终止的 supervisor* 的任务，数据处理就会失败。

2.1.2 supervisor 守护进程的工作方式

supervisor 守护进程等待 nimbus 分配任务后生成并监控 workers（JVM 进程）执行任务。supervisor 和 worker 都是运行在不同的 JVM 进程上，如果由 supervisor 拉起的一个 woker 进程因为错误（或者因为 Unix 终端的 kill-9 命令，Window 的 tskkill 命令强制结束）异常退出，supervisor 守护进程会尝试重新生成新的 worker 进程。

看到这里你可能想知道 Storm 的有保障传输机制如何适应其容错模型。如果一个 worker 甚至整个 supervisor 节点都故障了，Storm 怎么保障出错时正在处理的 tuples 的传输？

答案就在 Storm 的 tuple 锚定和应答确认机制中。当打开了可靠传输的选项，传输到故障节点上的 tuples 将不会收到应答确认，spout 会因为超时而重新发射原始 tuple。这样的过程会一直重复直到 topology 从故障中恢复开始正常处理数据。

2.1.3 Apache ZooKeeper 简介

ZooKeeper 使用一个简单的操作原语集合和分组服务，在分布式环境下提供了集中式的信息维护管理服务。它是一种简单但功能强大的分布式同步机制，允许客户端的应用程序监控或者订阅数据集中的部分数据，当数据产生，更新或者修改时，客户端都会收到通知。使用常见的 ZooKeeper 模式或方法，开发者可以实现分布式计算所需要的很多种机制，比如 Leader 选举，分布式锁和队列。

Storm 主要使用 ZooKeeper 来协调一个集群中的状态信息，比如任务的分配情况，worker 的状态，supervisor 之间的 nimbus 的拓扑度量。nimbus 和 supervisor 节点之间的通信主要是结合 ZooKeeper 的状态变更通知和监控通知来处理的。

Storm 对 ZooKeeper 的使用相对比较轻量化，不会造成很重的资源负担。对于重量级的数据传输操作，比如发布 topology 时传输 jar 包，Storm 依赖 Thirft 进行通信。我们将会

看到，topology 组件之间的数据传输（最影响效率的地方）是在底层进行的，并且经过了性能优化。

2.1.4　Storm 的 DRPC 服务工作机制

Storm 应用中的一个常见模式期望将 Storm 的并发性和分布式计算能力应用到“请求 - 响应”范式中。一个客户端进程或者应用提交了一个请求并同步地等待响应。这样的范式可能看起来和典型 topology 的高异步性、长时间运行的特点恰恰相反，Storm 具有事务处理的特性来实现这种应用场景，如图 2-2 所示。

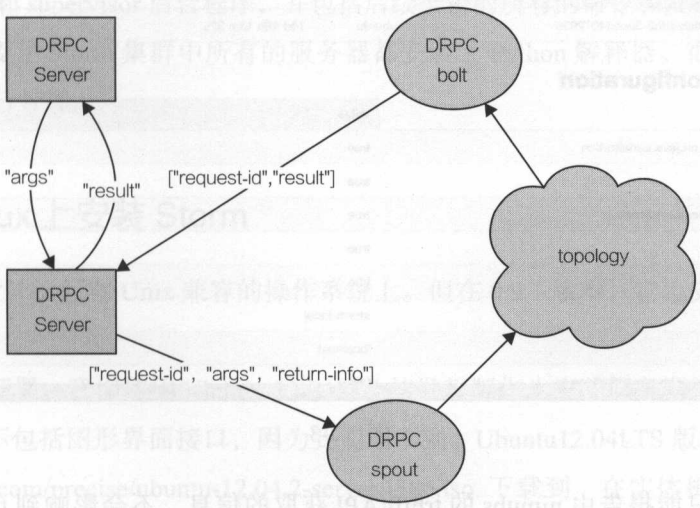


图　2-2

为了实现这个功能，Storm 将额外的服务（Storm DRPC）以及 spout 和 bolt 整合在一起工作，提供了可扩展的分布式 RPC 能力。

DRPC 功能是完全是可选的，当 Storm 集群中的应用有使用这个功能时，DRPC 服务节点才是必须的。

2.1.5　Storm UI

Storm UI 也是可选功能，非常有用，会提供一个基于 Web 的 GUI 来监控 Storm 集群，对正在运行的 topology 有一定的管理功能。Storm UI 提供了已经发布的 topology 的统计信息，对监控 Storm 集群的运转和 topology 的功能有很大帮助，如图 2-3 所示。

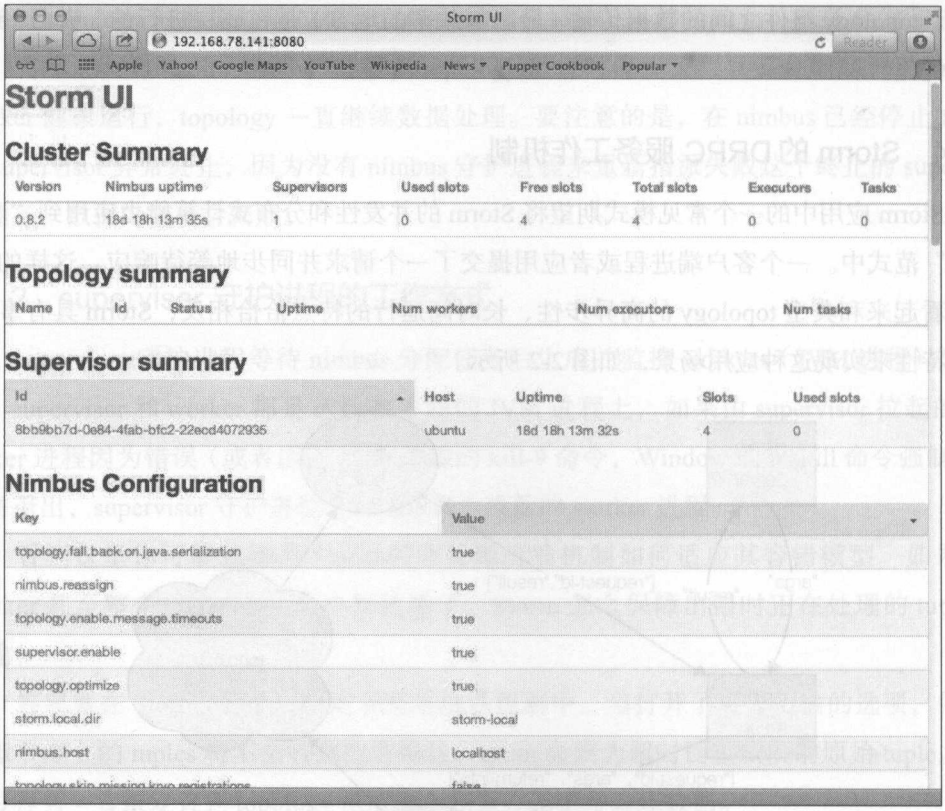


图 2-3

Storm UI 只能报告由 nimubs 的 trhift API 获取的信息，不会影响到 topology 上其他功能。Storm UI 可以随时开关而不影响任何 topology 的运行，在那里它完全是无状态的。它还可以用配置来进行一些简单的管理功能，如开启、停止、暂停和重新均衡负载 topology。

2.2 Storm 技术栈简介

在安装 Storm 之前，我们来看看 Storm 和 topology 是基于哪些技术建立的。

2.2.1 Java 和 Clojure

Storm 运行在 Java 虚拟机上，大部分是使用 Java 和 Clojure 进行开发的。Storm 的主

要接口都是通过 Java 语言定制的，Storm 使用 Python 实现了可执行程序。除了这些程序，由于 Java 使用了 Apache Thrift 接口，Java 还友好的兼容多种其他语言。

Storm 的组件（spout 和 bot）实际上可以使用任何当前服务器安装支持的语言进行开发。JVM 虚拟机支持的语言可以原生的执行，其他语言的实现需要通过 JNI 和 Storm 的多语言协议来实现。

2.2.2　Python

所有 Storm 的后台程序和管理命令都是使用单独一个可执行 Python 文件来启动。这包括了 nimbus 和 supervisor 后台程序，并包括后续会讲的所有的命令和发布管理命令。这样做的原因主要是 Storm 集群中所有的服务器都安装了 Python 解释器。很多工作站也使用 Python 来进行管理。

2.3　在 Linux 上安装 Storm

Storm 是设计运行在 Unix 兼容的操作系统上。但在 0.9.1 版本，它也支持在 Windows 机器上部署。

为了简化部署，我们使用 Ubuntu 12.04LTS 的发行版作为安装服务器。将会使用服务器版本，默认不包括图形界面接口，因为我们用不到。Ubuntu12.04LTS 版本可以在 http://releases.ubuntu.com/precise/ubuntu-12.04.2-server-i386.iso. 下载到。在实体机和虚拟机上安装 ubuntu 都是非常方便的。出于学习和开发的目的，你会发现在虚拟机里进行部署更加方便，尤其是手头没有那么多实体机的情况。

OSX、Linux、Windows 都有着对应的虚拟机软件。我们建议从下面集中软件中选择一个：

- VMWare（OSX、Linux、Windows）

这个软件是需要付费的，可以在 http://www.vmware.com 获得。

- VirtualBox（OSX、Linux、Windows）

这个软件是免费的，可以在 https://www.virtualbox.org 下载

- Parallels Desktop（OSX）

这个软件也是需要付费的，可以在 http://www.parallels.com 下载。

2.3.1 安装基础操作系统

可以从 Ubuntu 的安装盘或镜像启动，遵循屏幕上的指令选择基础安装。在 Packege Selection 在屏幕上出现的时候，选取安装 OpenSSH 服务的选项。这个安装包允许你从远程 ssh 登录上服务器。在其他安装界面上，都选择默认选项就可以了，除非是为了兼容硬件修改选项。

默认情况下，进入 Ubuntu 的初始化用户具有管理权限（sudo）。如果你要重新建一个账户，记得给账户赋予管理权限。

2.3.2 安装 Java

首先安装 JVM。Storm 可以兼容 1.6 和 1.7 版本 OpenJDK 和 Oracle 发布的 JVM。本例中，我们首先更新系统，然后安装 1.6 版本的 JDK。

```
sudo apt-get update
sudo apt-get --yes install openjdk-6-jdk
```

2.3.3 安装 ZooKeeper

在我们的单节点伪集群中，需要安装一个 ZooKeeper 实例就够了。Storm 需要 3.3.X 版本的 ZooKeeper。调用一次下述命令即可：

```
sudo apt-get --yes install zookeeper=3.3.5* zookeeperd=3.3.5*
```

这个命令会安装二进制版本的 ZooKeeper，并且会生成一个服务的脚本来启动和关闭它。还会生成一个 cron 工作来定期清理过期的 ZooKeeper 传输记录和快照。如果不按默认设置来清理，ZooKeeper 很快就会占用大量硬盘使用空间。

2.3.4 安装 Storm

Storm 二进制发行版本可以在 Storm 官网（http://storm.incubator.apache.org）下载到。二进制版本的安装文件夹布局更适合在生产系统中部署。我们会做一些修改，使之更贴近 UNIX 的风格（比如日志搭在 /var/log 下而不是搭在 Storm 的主目录）。

首先新建 Storm 的用户和用户组。这可以避免 Storm 进程以默认或者 root 权限启动：

```
sudo groupadd storm
sudo useradd --gid storm --home-dir /home/
storm --create-home --shell /bin/bash storm
```

然后下载和安装 Storm 的发行版本。我们将 Storm 安装在 /usr/share 目录下并且生成一个版本无关的软连接到 /usr/share/storm 目录下。这种方式便于将来升级版本，或回滚版本，只需要重新建立一个软连接即可。我们在将 Storm 的可执行程序软连接到 /usr/bin/storm 目录下：

```
sudo wget [storm download URL]
sudo unzip -o apache-storm-0.9.1-incubating.zip -d /usr/share/
sudo ln -s /usr/share/apache-storm-0.9.1-incubating /usr/share/storm
sudo ln -s /usr/share/storm/bin/storm /usr/bin/storm
```

默认的，Storm 会将日志写在 $STORM_HOME/logs，而不像其他 Unix 程序一样将日志写在 /var/log 下。执行下面的命令，在 /var/log 生成 Storm 转述的 log 目录，并将 Storm 配置指向那里：

```
sudo mkdir /var/log/storm
sudo chown storm:storm /var/log/storm

sudo sed -i 's/${storm.home}\/logs/\/var\/log\/storm/
  g' /usr/share/storm/log4j/storm.log.properties
```

最后，将 Storm 的默认配置文件移到 /etc/storm 下，建立软连接以便 Storm 程序可以找到它：

```
sudo mkdir /etc/storm
sudo chown storm:storm /etc/storm
sudo mv /usr/share/storm/conf/storm.yaml /etc/storm/
sudo ln -s /etc/storm/storm.yaml /usr/share/storm/conf/storm.yaml
```

Storm 安装完毕后，可以对 Storm 守护进程的程序进行配置，使其异常之后可以自动恢复。

2.3.5　运行 Storm 守护进程

所有的 Storm 守护进程都是设计为快速失败的，也就是一旦遇到了任何异常错误进程将会终止。这样使得单独的组件可以安全地结束，并且在不影响系统其他部分的情况下恢复。

这意味着，Storm 的守护进程无论在什么时候异常终止，都需要立即重启。这个技术称为在监督（supervision）下运行进程。幸运的是，有很多种技术供选择。实际上，ZooKeeper 也是一个快速失败的系统，在 Debian 发行版中特有的基于 upstart 的 init 脚本给 ZooKeeper 提供了这项功能。如果 ZooKeeper 进程在任意时间退出，upstart 会保证重启

进程恢复集群。

Debian 的 upstart 系统非常适合这种场景。在其他 Linux 发行版本还有其他的选择。为了简单起见，我们使用大部分 Linux 发行版都有的 supervisor 包来实现这个功能。不巧的是，supervisor 的名字和 Storm 的 supervisor 后台程序冲突了。为了表明区别，我们将非 Storm 的 supervisior 后台程序叫做 supervisord（添加了一个 d），但后面代码示例中，还是会用没有 d 的名字来执行命令。

在基于 Debian 的 Linux 发行版本中，supervisord 包命名为 supervisor，其在其他发行版，如 RED HAT 中，使用 supervisord 的名字，执行下述命令：

```
sudo apt-get --yes install supervisor
```

这条命令会安装和启动 supervisord 服务，服务的主要配置在 /etc/supervisor/supervisord 中。Supervisord 服务会自动包含所有 /etc/supervisord/conf.d/ 目录下的 *.conf 文件，我们将在这里放置运行 Storm 后台进程需要的 config 文件。

每个要在 Supervisord 监督下运行的命令，都需要创建一个文件，包含下列内容：

- 每个被监督命令要配置一个在 supervisord 配置中唯一的名字。
- 启动的命令
- 启动时的工作目录
- 当一个命令或者服务终止时，是否要拉起。对于快速失败服务，这项配置永远是 true。

切换到 Storm 命令时使用的目录。这里，我们使用 Storm 用户来执行所有守护进程

建立下述三个文件使得 Storm 守护进程由 supervisord 自动拉起（在遇到异常终止的事件时也会重启）。

- /etc/supervisord/conf.d/storm-nimbus.conf 文件内容是：

```
[program:storm-nimbus]
command=storm nimbus
directory=/home/storm
autorestart=true
user=storm
```

- /etc/supervisord/conf.d/storm-supervisor.conf 文件内容是：

```
[program:storm-supervisor]
command=storm supervisor
directory=/home/storm
autorestart=true
user=storm
```

- /etc/supervisord/conf.d/storm-ui.conf 文件内容是：

```
[program:storm-ui]
command=storm ui
directory=/home/storm
autorestart=true
user=storm
```

当文件生成后，通过下述命令终止启动 supervisord 服务：

```
sudo /etc/init.d/supervisor stop
sudo /etc/init.d/supervisor start
```

supervisord 会载入新的配置文件并且启动 Storm 的守护进程。片刻后 Storm 服务就会启动，可以通过在浏览器里访问下述 URL 来判断 Storm 伪集群是否已经启动运行（使用实际机器的 IP 或者 hostname 来替换 URL 中的 localhost 字段）：

```
http://localhost:8080
```

这里提及了 Storm UI 图形接口。它表明集群正在运行，包含了一个 supervisor 和四个 worker 槽位，没有 topology 在集群中运行（稍后将会将一个 topology 发布到集群里）。

如果 Storm UI 因为某些原因未能启动起来，或者看不到集群中活跃的 supervisor，可以从下述位置查看错误日志：

- Storm UI：在 /var/log/storm 中查看 ui.log 文件。
- Nimbus：在 /var/log/storm 中查看 nimbus.log 文件。
- Supervisor：在 /var/log/storm 查看 supervisor.log 文件。

迄今为止我们都在使用 Storm 的默认配置，使用 localhost 作为集群中 ZooKeeper、Nimbus 等服务的主机名参数。在一个机器上构建单节点伪集群时是可行的，但是实际搭建一个多节点的集群需要重新定义一些默认配置选项。

下来，我们将介绍 Storm 提供的不同的配置选项，以及这些配置项如何影响集群和 topology 的行为。

2.3.6　配置 Storm

Storm 的配置由一些 YAML 格式的属性组成。当一个 Storm 的守护进程启动时，会加载默认属性，然后加载 $STORM_HOME/conf 目录下的 storm.yaml（已经软连接到 /etc/storm/storm.yaml）文件，然后使用该文件中的值替换默认值。

下面列出了 storm.yaml 配置文件中必须要重新定义的几项内容：

```
# List of hosts in the zookeeper cluster
storm.zookeeper.servers:
    - "localhost"

# hostname of the nimbus node
nimbus.host: "localhost"

# supervisor worker ports
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703

# where nimbus and supervisors should store state data
storm.local.dir: "/home/storm"

# List of hosts that are Storm DRPC servers (optional)
# drpc.servers:
#     - "localhost"
```

2.3.7 必需的配置项

下述配置项是生产环境的多节点 Storm 集群的必选配置：

- storm.zookeeper.serviers：这项配置列出了 ZooKeeper 集群的主机名称。因为我们在一台机器上运行单节点的 ZooKeeper，并且 Storm 的其他守护程序都在同一台机器上，可以使用默认值的 localhost。
- nimbus.host：指定了集群中 nimbus 的节点。worker 需要从这项配置知道集群的主节点在哪里，用来下载 topology 的 jar 包和配置选项。
- supervisor.slots.pors：这个配置控制每个 supervisor 节点运行多少个 worker 进程。这个配置定义为 worker 监听端口的列表，监听端口的个数控制了 supervisor 节点上有多少个 worker 的插槽。例如，如果我们有个集群中有三个 supervisor 节点，每个节点配置了三个监听端口，整个集群就有九个 worker 插槽（3 × 3=9）。默认的，Storm 使用 6700~6703 端口，每个 supervisor 节点上有 4 个 worker 插槽。

- storm.local.dir：nimbus 和 supervisor 守护进程都会存储一些短暂的状态信息，比如 JAR 报和 woker 需要的配置文件。这个配置项决定了 nimbus 和 supervisor 将信息存储在哪里。这个指定的目录必须已经存在，并且 storm 的启动用户要有合适的操作权限，可以读或者写这个目录。这个目录下的内容必须在集群运行的过程中一直保存，所以要避免使用 /tmp 目录作为这个这个配置项，因为重启后 /tmp 目录的内容会丢失。

2.3.8 可选配置项

一个运营中的 Storm 集群除了上述必选配置项之外，还有一系列可选配置项，可以按照需要进行重新定义。Storm 配置选项使用点号分隔的命名规范，用前缀来识别配置分类；如表 2-1 所示。

表 2-1

前 缀	分 类	前 缀	分 类
storm.*	通用配置	supervisor.*	Supervisor 配置
nimbus.*	Nimbus 配置	worker.*	Worker 配置
ui.*	Storm UI 配置	zmq.*	ZeroMQ 配置
drpc.*	DRPC 服务配置	topology.*	Topology 配置

如果需要查看完整的默认配置，可以在 Storm 的源码中查看 defaults.yaml 文件（https://github.com/nathanmarz/storm/blob/master/conf/defaults.yaml）。其他几个经常需要重新定义的配置如下：

- nimbus.childopts(default:“-Xms1024m”)：这项 JVM 的配置会添加在启动 nimbus 守护进程的 Java 命令行中。
- ui.port(default:8080)：这项配置指定了 Storm UI 的 Web 服务器监听的端口。
- ui.childopts(default:“-Xms1024m”)：这项 JVM 配置会添加在 Storm UI 服务启动的 Java 命令行中。
- supervisor.childopts(default:“-Xms768m”)：这个 JVM 选项会添加在 Supervisor 启动的 Java 命令行中。
- worker.childopts(default:“-Xms768m”)：这个 JVM 选项会添加在启动 worker 进程的 Java 命令行中。

- topology.message.timeout.secs(default:30)：这个配置项设定了一个 tuple 树需要应答最大时间秒数限制，超过这个时间的认为已经执行失败（超时）。这个值设置得太小可能会导致 tuple 反复重新发送。当这个选项生效时，spout 必须设定来发送锚定的 tuple。
- topology.max.spout.pending(default:null)：在默认值 null 的时候，每当 spout 产生了新的 tuple，Storm 会立即将 tuple 向后端数据流发送。由于下游 bolt 执行可能有延迟，默认的数据发送行为可能导致 topology 过载，从而导致消息处理超时。将本选项设置为非 null 大于 0 的数字时，Storm 会暂停发送 tuple 到数据流直到发送出去的 tuple 小于这个数字，起到了对 spout 限速的作用。这项配置和 topology.message.timeout.secs 一起，是调节 topology 性能的最重要的两个参数。
- topology.enable.message.timeouts(default:ture)：这个选项用来设定锚定的 tuple 的超时时间。如果设置为 false，则锚定的 tuple 不会超时。谨慎使用这个选项，将本项配置设置为 flase 之前，考虑改变 topology.message.timeout.secs。本项配置生效后，spout 必须配置为发送锚定 tuple。

2.3.9 Storm 可执行程序

Storm 执行程序是一个多用途的命令行程序，可以用来启动所有的守护进程，执行 topology 管理操作，比如说部署一个新的 topology 到集群，或者使用本地模式在开发测试阶段执行一个 topology。

Storm 命令的基础命令如下：

```
storm [command] [arguments...]
```

2.3.10 在工作站上安装 Storm 可执行程序

为了在连接的远程集群上执行 Storm 命令，需要在本地安装 Storm 发行版。在工作站上安装 Storm 发行版很简单，只需要解压缩 Storm 发行版的压缩包，并且添加 Stom 的 bin 目录（$STORM_HOME/bin）到 PATH 环境变量中。下一步在 ~/.storm/ 目录下新建 storm.yaml 文件，文件内容只有一行，告诉 storm 在哪里可以找到需要连接集群的 nimbus 服务：

示例：~/.storm/storm.yaml file.

```
Sample: ~/.storm/storm.yaml file.
nimbus.host: "nimbus01."
```

为了让一个 Storm 集群正确执行，必须正确配置主机名的 IP 地址解析，无论是通过 DNS 系统还是通过 /etc/hosts 文件。

在 Storm 配置中，也可以直接使用 IP 来代替主机名，使用 DNS 系统更好。

2.3.11　守护进程命令

Storm 的守护进程用来启动 Storm 服务，应该在被监督模式下启动，这样，程序如果有异常失败，服务就可以重启。在启动时，Storm 守护程序从 $STORM_HOME/conf/storm.yaml 中读取配置。这个配置文件中的任何配置项都会重新定义内置的默认配置。

Nimbus

用法：storm nimbus

这个命令启动 nimbus 守护进程

Supervisor

用法：storm supervisor

这个命令启动 supervisor 进程

UI

用法：storm ui

这个命令启动 Storm UI 的守护进程，提供了一个 web 的 UI 界面用来监控 Storm 集群。

DRPC

用法：storm drpc

这个命令启动一个 DRPC 服务守护进程

2.3.12　管理命令

Storm 的管理命令用来发布和管理集群中的 topology。管理命令通常在 Storm 集群外部的工作站来执行，但不是必需的。和 nimbus Thrift API 通信需要知道 nimbus 节点的主

机名。管理命令在 ~/.storm/storm.yaml 文件中查找配置，Storm 的 jar 包会添加到 Java 的 classpath 中。配置中唯一需要的参数是 nimbus 节点的主机名：

```
nimbus.host: "nimbus01"
```

Jar

用法：storm jar topology_jar topology_class[arguments...]

jar 命令用来向集群提交 topology。它会使用指定的参数运行 topology_class 中的 main() 方法，同时上传 topology_jar 文件到 nimbus 以分发到整个集群。提交后，Storm 集群会激活并且开始运行 topology。

topology 类中的 main() 方法需要调用 StormSubmitter.submitTopology() 方法，并且为 topology 提供集群内唯一的名称。如果集群中已经有一个同名的 topology，jar 命令会执行失败。通常会通过命令行参数的方式指定 topology 的名称，这样 topology 就可以在提交执行的时候再命名。

Kill

用法：storm kill topology_name[-w wait_time]

Kill 命令用来关闭已经部署的 topology。这个命令使用 topology_name 来关闭 topology。首先，Storm 在 topology.message.timeout.secs 的时间后使 toplogy 的 spout 取消激活，这样已经发出去的 tuple 就可以执行完毕。然后停止 worker 进程，并且尝试清理所有存储的状态信息。特别是，通过 -w 参数，可以使用特定的时间间隔覆盖 topology.message.timeout.secs 参数。Kill 命令也可以用在 Storm UI 上进行操作。

Deactivate

用法：storm deactivate topology_name

Deactivate 命令告诉 Storm 停止特定的 topology 的 spout 发送 tuple。topology 可以在 Storm UI 上进行取消激活的操作。

Activate

用法：storm activate topology_name

Activate 命令告诉 Storm 重新恢复特定 topology 的 spout 发送 tuple，topology 可以在

Storm UI 中重新激活操作。

Rebalance

用法：storm rebalance topology_name[-w wait_time][-n worker_count][-e component_name=executer_count]...

rebalance 命令指示 Storm 在集群的 worker 之间重新平均地分派任务，不需要关闭或者重新提交现有的 topology。例如，当一个新的 supervisor 节点添加到一个集群中时，就会需要执行这个命令，因为现有的 topology 是不会将任务分配到新节点的 worker 上的。

rebalance 命令还可以分别使用 -n 和 -e 参数来修改为 topology 分配的 worker 的个数以及每个 task 分配的 executor 的个数。

当执行 rebalance 命令时，Storm 会先取消激活 topology，等待配置的时间使剩余的 tuple 完成处理，然后在 supervisor 节点中均匀地重新分配 worker。重新平衡后，Storm 会将 topology 恢复到之前的激活状态（意思是，如果是已经激活的 topology，Storm 会重新激活它，反之亦然）。

下述的例子会等待 15 秒后重新平衡 wordcount-topology topology，指定 5 个 worker，比如去设置 sentence-spout 和 split-bolt 使用 4 个和 8 个 executor 线程：

```
storm rebalance wordcount-topology -w 15 -n 5 -e
    sentence-spout=4 -e split-bolt=8
```

Remoteconfvalue

用法：storm remoteconfvalue conf-name

Remoteconfvalue 命令用来查看远程集群中的配置参数值。注意，用这个命令看到的是整个集群的公共配置，看不到单独 topology 中覆盖的特殊配置。

2.3.13　本地调试 / 开发命令

Storm 的本地命令是用来进行调试和测试用的。和管理命令类似，Storm 的调试命令也会读取 ~/.storm/storm.yaml 文件并且使用其值来覆盖 Storm 内置的默认值。

REPL

使用：storm repl

Repl 命令会使用 Storm 的本地 classpath 打开一个 Clojure REPL 会话。

Classpath

使用：storm classpath

Classpath 命令打印 Storm client 使用的 classpath 值。

Localconfvalue

使用：storm localconfvalue conf-name

Localconfvalue 命令在整合 ~/.storm/storm.yaml 和 Storm 内置默认值后的配置中查找特定配置项的值。

2.4 把 toplogy 提交到集群中

现在我们已经有一个正在运行的 Storm 集群了，让我们回顾一下前面说的单词计数的例子，然后使之能够像本地模式一样运行在集群环境。前面的例子使用 LocalCluster 类将 topology 运行在本地模式：

```
LocalCluster cluster = new LocalCluster();
            cluster.submitTopology(TOPOLOGY_NAME, config, builder.
createTopology());
```

提交一个 topology 到远程集群非常简单，只需要使用 StormSubmitter 类中同样的方法和名称：

```
StormSubmitter.submitTopology(TOPOLOGY_NAME, config,
  builder.createTopology());
```

当开发一个 Storm 的 topology 时，通常不想在本地 / 远程集群模式之间切换部署时修改代码和重新编译。标准的方法是使用一个 if/else 的条件块，使用命令行参数来决定使用哪种模式，当命令行不带参数时，使用本地模式，反之当使用 topology 名称做参数时，使用远程集群模式，如下所示：

```
public class WordCountTopology {

    private static final String SENTENCE_SPOUT_ID =
    "sentence-spout";
    private static final String SPLIT_BOLT_ID = "split-bolt";
    private static final String COUNT_BOLT_ID = "count-bolt";
    private static final String REPORT_BOLT_ID = "report-bolt";
    private static final String TOPOLOGY_NAME =
```

```
    "word-count-topology";

    public static void main(String[] args) throws Exception {

        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout(SENTENCE_SPOUT_ID, spout, 2);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2)
                .setNumTasks(4)
                .shuffleGrouping(SENTENCE_SPOUT_ID);
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt(COUNT_BOLT_ID, countBolt, 4)
                .fieldsGrouping(SPLIT_BOLT_ID,
                new Fields("word"));
        // WordCountBolt --> ReportBolt
        builder.setBolt(REPORT_BOLT_ID, reportBolt)
                .globalGrouping(COUNT_BOLT_ID);

        Config config = new Config();
        config.setNumWorkers(2);

        if(args.length == 0){
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology(TOPOLOGY_NAME, config,
            builder.createTopology());
            waitForSeconds(10);
            cluster.killTopology(TOPOLOGY_NAME);
            cluster.shutdown();
        } else{
            StormSubmitter.submitTopology(args[0],
            config, builder.createTopology());
        }
    }
}
```

为了更新单词计数程序到运行中的集群，首先在第2章的代码目录执行Maven的编译命令：

```
mvn clean install
```

然后，执行storm jar命令来发布topology：

```
storm jar ./target/Chapter1-1.0-SNAPSHOT.jar
  storm.blueprints.chapter1.WordCountTopology wordcount-topology
```

当命令执行完毕时，你应该看到 topology 在 Storm UI 里已经激活，并且可以点击 topology 的名称来查看详情和 topology 的统计信息，如图 2-4 所示。

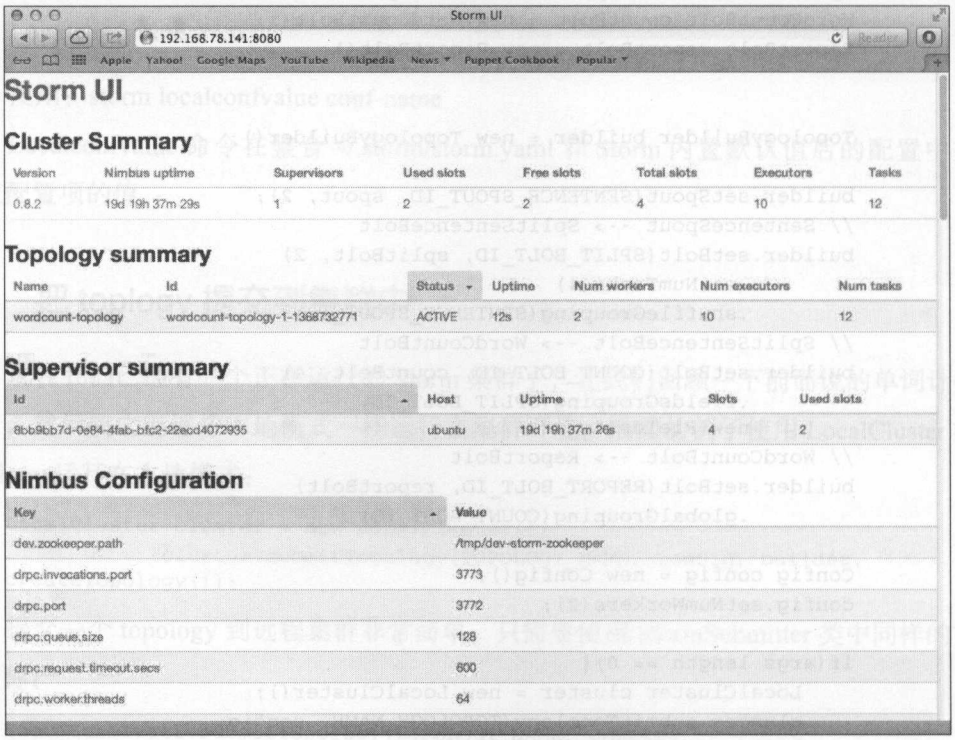


图 2-4

2.5 自动化集群配置

迄今为止，我们已经在命令行模式下手工配置了单节点伪集群。这种方法应对小规模集群当然是行得通，但当集群规模不断增加时，这种方式就变得难以维护了。考虑到配置十个，甚至成百上千节点的场景。配置任务可以通过脚本来自动执行，但即使是基于脚本的自动化解决方案在扩展性上也是有问题的。

幸运的是，已有一些有效的技术来帮助解决管理大规模服务器集群的配置和定制问题。Chef 和 Puppet 都提供了声明式的配置方式，来定义机器的状态（意味着机器安装了

哪些软件包，以及如何配置的）和分类（例如，一个 Apache Web Server 类机器需要安装 Apache 的 httpd 程序）。

服务器定制和配置过程的自动化是一个非常大的主题，超出了本书的范畴。我们的目的是使用 Puppet 中的一部分功能，介绍其基本的概念，同时鼓励大家进一步深入了解。

2.6　Puppet 的快速入门

Puppet(https://puppetlabs.com) 是一个 IT 自动化框架，用来帮助管理员管理大规模的网络设施资源，使用灵活的、声明式的实现方式来进行 IT 自动化管理。

Puppet 的核心是 manifest 的概念，它描述了一个设施资源对预期的声明（state）。在 Puppet 中，声明包括了以下方面：

- 安装了哪些软件包。
- 运行着哪些服务，不运行哪些服务。
- 软件配置详情。

2.6.1　Puppet manifest 文件

Puppet 使用了基于 Ruby 的 DSL（Domain Specific Language）在一组 manifest 文件中来描述系统配置。例如，描述 ZooKeeper 的 Puppet manifest 文件如下：

```
package { 'zookeeper':
    ensure => "3.3.5*",
}

package { 'zookeeperd':
    ensure => "3.3.5*",
    require => Package["zookeeper"],
}

service { 'zookeeperd':
    ensure => 'running',
    require => Package["zookeeperd"],
}
```

这个最简单的 manifest 例子表明 ZooKeeper 作为一个服务已经安装在机器上，并且服务是运行的。第一个 package 的代码块告诉 Puppet 使用操作系统的包管理程序（例如，Ubuntu/debian 的 apt-get，Ret Hat 的 yum 等）来安装 3.3.5 版本的 ZooKeeper 软件包。第

二个 package 代码块保证 zookeeperd 软件包已经安装。最后 service 代码块告诉 Puppet 需要保证 zookeeperd 的系统服务正在运行，这个服务依赖于 zookeeperd 软件包已经安装。

为了演示 Puppet manifest 文件如何转化为安装软件和系统声明，让我们安装 Puppet 并且使用前面的例子来安装和启动 zookeeperd 服务。

为了获取最新版的 Puppet，需要将 apt-get 配置使用 Puppet 实验室的软件源。执行下述命令来安装最新版的 Puppet：

```
wget http://apt.puppetlabs.com/puppetlabs-release-precise.deb
sudo dpkg -i puppetlabs-release-precise.deb
sudo apt-get update
```

然后，保存前面列出的实例 manifest 文件，命名为 init.pp。使用 Puppet 来应用这个 manifest：

```
sudo puppet apply init.pp
```

当命令执行完后，查看 zookeeper 服务是否正在运行：

```
service zookeeper status
```

如果我们手动停止了 zookeeper 服务并且重新运行 Puppet apply 命令，Puppet 不会重复执行安装过程，但是会重启服务，因为在 manifest 文件中定义的声明中，服务是运行的。

2.6.2 Puppet 类和模块

使用独立的 Puppet manifest 文件很容易定义资源的声明，但这种实现方法在需要管理的资源增多时变得越来越不方便。所幸是 Puppet 有类和模块的概念，可以用来更好地组织和隔离特定的配置详情。

考虑到 Storm 的场景，我们有多个节点的类。举例来说，一个 Storm 集群会有 nimbus 节点，supervisor 节点或者同时兼具两种服务的节点。Puppet 类和模块提供了一种区分不同配置角色的方法，可以混合搭配来方便地定义身具多种角色的资源。

举例说明这个功能，回顾上面安装 zookeeper 包的 manifest 文件，将它重新定义为一个类，这样包含多个类的 manifest 文件就可以引用这个类：

```
class zookeeper {

    include 'jdk'

    package { 'zookeeper':
```

```
        ensure => "3.3.5*",
    }
    package { 'zookeeperd':
        ensure => "3.3.5*",
        require => Package["zookeeper"],
    }

    service { 'zookeeperd':
        ensure => 'running',
        require => Package["zookeeperd"],
    }
}
```

在前面的例子里，我们将 zookeeper 的 manifest 文件重新定义为 puppet 的类。这个类可以在其他类或 manifest 文件中使用。第二行中，zookeeper 类包含了另外一个类，jdk，引用这个类的定义，表示它声明一个机器必须具备 Java JDK。

2.6.3　Puppet 模板

Puppet 还利用 Ruby ERB 模板系统来定义各种文件的模板，在 Puppet 生成 manifest 时使用。当 Puppet 运行时，会替换 Puppet ERB 模板中 Ruby 表达式和构造式。ERB 模板中的 Ruby 语句对 manifest 文件中的 Puppet 变量和定义有完全访问的权限。

下述 Puppet 文件声明是用来生成 storm.yaml 配置文件的：

```
file { "storm-etc-config":
    path => "/etc/storm/storm.yaml",
    ensure => file,
    content => template("storm/storm.yaml.erb"),
    require => [File['storm-etc-config-dir'],
      File['storm-share-symlink']],
}
```

这个声明告诉 Puppet，从 storm.yaml.erb 模板中，在 /etc/storm 下新建 storm.yaml 文件：

```
storm.zookeeper.servers:
<% @zookeeper_hosts.each do |host| -%>
    - <%= host %>
<% end -%>

nimbus.host: <%= @nimbus_host %>

storm.local.dir: <%= @storm_local_dir %>

<% if @supervisor_ports != 'none' %>
supervisor.slots.ports:
```

```
<% @supervisor_ports.each do |port| -%>
    - <%= port %>
<% end -%>
<% end %>

<% if @drpc_servers != 'none' %>
<% @drpc_servers.each do |drpc| -%>
    - <%= drpc %>
<% end -%>
<% end %>
```

模板中的条件逻辑和变量扩展使我们在多种环境下可以使用同一个模板文件。举例来说，如果环境中不需要 Storm DRPC 服务，则生成的 storm.yaml 文件中就会忽略掉 drpc.service 的部分。

2.6.4 使用 Puppet Hiera 来管理环境

我们已经简要介绍了 Puppet manifest、类和模板的概念。你这时候可能会好奇，如何在一个 puppet 的类或 manifest 文件中定义变量。在 puppet 类或 manifest 文件中定义变量非常简单，在 manifest 文件和类中像下面这样定义即可：

```
$java_version = "1.6.0"
```

定义后，java_sersion 变量就可以在类或 manifest 以及 ERB 模板中使用了；但是，这也带来了不易复用的问题。如果我们将版本号等信息写死在代码里，就会因固定的变量值限制了这个类的复用效率。如果可以将变量扩展到可能潜在的各种变化值，配置管理会变得更容易维护。这就是 Hiera 的用武之地。

2.6.5 介绍 Hiera

Hiera 是一个最新版 Puppet 框架中集成的一个键值对的查找工具。Hiera 允许定义键值对分层结构（hierarchies，Hiera 的名字是缩写），在结构中，父节点定义的值可以被子节点定义的值代替。

例如，考虑我们在 Storm 集群中需要给一些机器定义配置参数的场景。所有的机器会使用一套通用的键值对，比如说使用的 Java 版本。所以我们将这些值定义在 common.yaml 中。从这里开始，开始出现分化。我们可能有单节点伪集群，也可能有多节点集群。我们会将环境相关的变量放在特殊的文件里，比如“single-node.yaml”和“cluster.

yaml”。

最后，我们会将机器相关的信息写在这种格式的文件中：“[hostname].yaml”，如图 2-5 所示。

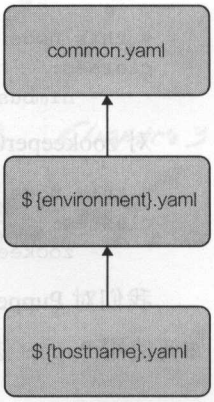


图　2-5

Puppet 集成的 Hiera 允许使用 Puppet 内置变量来正确的解析文件名称。

第 2 章的示例源代码目录说明了如何实现文件这种文件组织方式。一个典型的 common.yaml 文件可能定义全局属性对所有的机器通用，如下格式：

```
storm.version: apache-storm-0.9.1-incubating

# options are oracle-jdk, openjdk
jdk.vendor: openjdk
# options are 6, 7, 8
jdk.version: 7
```

在 environment 级别，可以区分单节点和集群配置的不同，cluster.yaml 可以像下面这样定义：

```
# hosts entries for name resolution (template params for /etc/hosts)
hosts:
    nimbus01: 192.168.1.10
    zookeeper01: 192.168.1.11
    supervisor01: 192.168.1.12
    supervisor02: 192.168.1.13
    supervisor04: 192.168.1.14


storm.nimbus.host: nimbus01

storm.zookeeper.servers:
    - zookeeper01

storm.supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
    - 6705
```

最后，是定义机器相关的参数配置，使用对应的 [hostname].yaml 文件，定义的 Puppet 类会针对特定节点。

比如 nimbus01.yaml，使用下列配置：

```
# this node only acts as a nimus node
classes:
    - nimbus
```

对 zookeeper01.yaml。使用下列代码：

```
# this node is strictly a zookeeper node
classes:
    - zookeeper
```

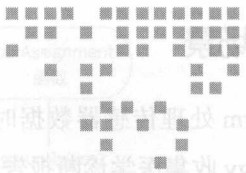
我们对 Puppet 和 Hiera 仅仅了解了皮毛。第 2 章的代码中有额外的例子和文档来说明如何使用 Puppet 进行自动部署和配置任务。

总结

在本章中，我们介绍了安装和配置单节点和多节点集群 Storm 的必须步骤。还介绍了 Storm 的守护进程和命令行工具用来管理和运行 topology。

最后，简要介绍了 Puppet 框架，演示如何使用该框架管理多节点环境下的配置。鼓励大家更深入研究下载的代码和文档。

下一章，我们将介绍 Trident，它是一个在 Storm 事务处理和状态管理基础上高级别的抽象技术。



第 3 章 Chapter 3

Trident 和传感器数据

在本章中，我们将介绍 Trident topology。Trident 在 Storm 上提供了高层抽象。Trident 抽象掉了事务处理和状态管理的细节。特别是，它可以让一批 tuple 进行离散的事务处理。此外，Trident 还提供了抽象操作，允许 topology 在数据上执行函数功能、过滤和聚合操作。

我们将使用传感器数据作为例子来更好地理解 Trident。通常情况下，传感器数据流会来自不同的位置。一些传统的例子包括天气和交通状况，这种模式扩展到了更大的数据源。比如，手机应用产生的众多事件信息。处理手机生成的事件流就是另一个传感器数据处理的实例。

传感器数据包括不同设备发射的事件，往往是无穷尽的数据流。这正是 Storm 最合适的一种应用场景。

本章包括以下主题：

- Trident topology
- Trident spout
- Trident 操作——filter 和 function
- Trident 聚合——Combiner 和 Reducer
- Trident 状态（state）

3.1 使用场景

在用 Storm 处理传感器数据时，为了更好地理解 Trident topology，我们实现了一个 Trident topolygy 收集医学诊断报告来判断是否有疾病暴发的实例。

这个 topology 会处理的医学诊断事件包括以下的信息：

Latitude	Longitude	Timestamp	Diagnosis Code(ICD9-CM)
39.9522	-75.1642	03/13/2013 at 3:30 PM	320.0(*Hemophilus meningitis*)
40.3588	-75.6269	03/13/2013 at 3:50 PM	324.0(*Intracranial abscess*)

每个事件包括事件发生时的全球定位系统（GPS）的位置坐标，经度和纬度使用十进制小数表示。事件还包括 ICD9-CM 编码，表示诊断结果，以及事件发生的时间戳。完整的 ICD9-CM 编码参见 http://www.icd9data.com/。

为了判断是否有疾病暴发，系统会按照地理位置来统计各种疾病代码在一段时间内出现的次数。为了简化例子，我们按城市划分诊断结果地理位置。实际系统会对地理位置做出更精细的划分。

另外，示例中会逐小时对诊断事件进行分组。实际系统会更倾向于使用滑动窗口，使用移动平均值来计算趋势。

最后，我们使用简单的阈值来判断是否有疾病暴发。如果某个小时事件发生的次数超过了阈值，系统会产生告警信息并且派遣应急人员。

为了维护历史记录，我们还需要将每个城市、小时、疾病的统计量持久化存储。

3.2 Trident topology

为了满足这些需求，我们需要在 topology 中对疾病的发生进行统计。使用标准的 Storm topology 进行统计会遇到难题，因为 tuple 可能重复发送，这会导致重复计数的问题。下一节将会看到，Trident 提供了操作原语来解决这个问题。

我们将使用的 topology，如图 3-1 所示。

上述 topology 的代码如下：

```
public class OutbreakDetectionTopology {

    public static StormTopology buildTopology() {
    TridentTopology topology = new TridentTopology();
```

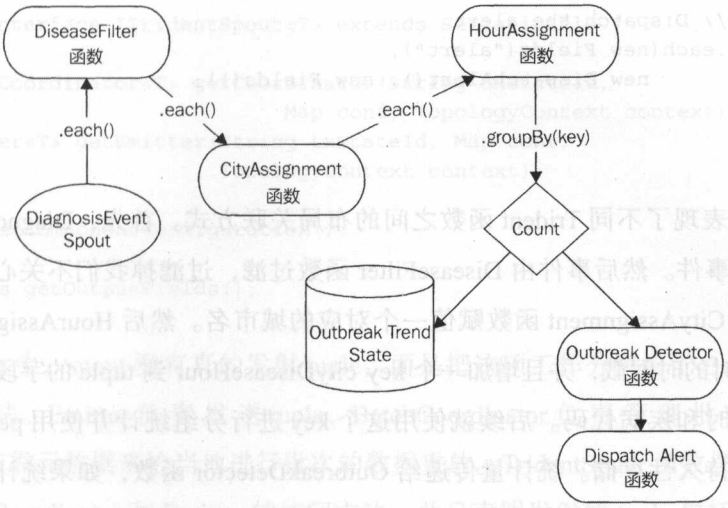


图　3-1

```
    DiagnosisEventSpout spout = new DiagnosisEventSpout();
    Stream inputStream = topology.newStream("event", spout);
    inputStream
    // Filter for critical events.
.each(new Fields("event"), new DiseaseFilter()))

        // Locate the closest city
        .each(new Fields("event"),
              new CityAssignment(), new Fields("city"))
        // Derive the hour segment
        .each(new Fields("event", "city"),
              new HourAssignment(), new Fields("hour",
              "cityDiseaseHour"))

        // Group occurrences in same city and hour
        .groupBy(new Fields("cityDiseaseHour"))

        // Count occurrences and persist the results.
        .persistentAggregate(new OutbreakTrendFactory(),
                             new Count(),
                             new Fields("count"))

        .newValuesStream()

        // Detect an outbreak
        .each(new Fields("cityDiseaseHour", "count"),
              new OutbreakDetector(), new Fields("alert"))
```

```
        // Dispatch the alert
        .each(new Fields("alert"),
              new DispatchAlert(), new Fields());

}
}
```

上述代码表现了不同 Trident 函数之间的布局关联方式。首先，DiagnosisEventSpout 函数发射疾病事件。然后事件由 DiseaseFilter 函数过滤，过滤掉我们不关心的疾病事件。之后，事件由 CityAssignment 函数赋值一个对应的城市名。然后 HourAssignment 函数赋值一个表示小时的时间戳，并且增加一个 key cityDiseaseHour 到 tuple 的字段中，这个 key 包括城市、小时和疾病代码。后续就使用这个 key 进行分组统计并使用 persistAggregate 函数对统计量持久性存储。统计量传递给 OutbreakDetector 函数，如果统计量超过阈值，OutbreakDetector 向后发送一个告警信息。最后 DispatchAlert 接收到告警信息，记录日志，并且结束流程。在后面，我们会深入了解每个步骤。

3.3 Trident spout

让我们先来看 topology 中的 spout。和 Storm 相比，Trident 引入了“数据批次”(batch) 的概念。不像 Storm 的 spout，Trident spout 必须成批地发送 tuple。

每个 batch 会分配一个唯一的事务标识符。spout 基于约定决定 batch 的组成方式。spout 有三种约定：非事务型 (non-transactional)、事务型 (transactional)、非透明型 (opaque)。

非事务型 spout 对 batch 的组成部分不提供保障，并且可能出现重复。两个不同的 batch 可能含有相同的 tuple。事务型 spout 保证 batch 是非重复的，并且 batch 总是包含相同的 tuple。非透明型 spout 保证数据是非重复的，但不能保证 batch 的内容是不变的。

表 3-1 描述了这些特性。

表 3-1

spout 类型	batch 可能有重复数据	batch 内容不会变化
非事务型	×	×
非透明型		×
事务型		

spout 接口如下面代码片段所示：

```
public interface ITridentSpout<T> extends Serializable {

    BatchCoordinator<T> getCoordinator(String txStateId,
                                       Map conf, TopologyContext context);
    Emitter<T> getEmitter(String txStateId, Map conf,
                          TopologyContext context);

    Map getComponentConfiguration();

    Fields getOutputFields();
}
```

在 Trident 中，spout 没有真的发射 tuple，而是把这项工作分解给了 BatchCoordinator 和 Emitter 方法。Emitter 负责发送 tuple，BatchCoordinator 负责管理批次和元数据，Emitter 需要依靠元数据来恰当地进行批次的数据重放。TridentSpout 函数仅仅是简单地提供了到 BatchCoordinator 和 Emitter 的访问方法，并且声明发射的 tuple 包括哪些字段。下面列出了示例中的 DiagnosisEventSpout 方法：

```
public class DiagnosisEventSpout implements ITridentSpout<Long> {
 private static final long serialVersionUID = 1L;
 SpoutOutputCollector collector;
 BatchCoordinator<Long> coordinator = new DefaultCoordinator();
 Emitter<Long> emitter = new DiagnosisEventEmitter();

 @Override
 public BatchCoordinator<Long> getCoordinator(
         String txStateId, Map conf, TopologyContext context) {
     return coordinator;
 }

 @Override
 public Emitter<Long> getEmitter(String txStateId, Map conf,
                                 TopologyContext context) {
     return emitter;
 }

 @Override
 public Map getComponentConfiguration() {
     return null;
 }

 @Override
 public Fields getOutputFields() {
     return new Fields("event");
 }
}
```

如上述代码中的 getOutputFields() 方法所示，在我们的实例 topology 中，spout 发射一个字段 event，值是一个 DiagnosisEvent 类。

BatchCoordinator 类实现下述接口：

```
public interface BatchCoordinator<X> {
   X initializeTransaction(long txid, X prevMetadata);
   void success(long txid);
   boolean isReady(long txid);
   void close();
}
```

BatchCoordinator 是一个泛型类。这个泛型类是重放一个 batch 所需要的元数据。在本例中，spout 发送随机事件，因此元数据可以忽略。实际系统中，元数据可能包含组成了这个 batch 的消息或者对象的标识符。通过这个信息，非透明型和事务型 spout 可以实现约定，确保 batch 的内容不出现重复，在事务型 spout 中，batch 的内容不会出现变化。

BatchCoordinator 类作为一个 Storm Bolt 运行在一个单线程中。Storm 会在 ZooKeeper 中持久化存储这个元数据。当事务处理完成时会通知到对应的 coordinator。

在我们的例子中，没有做特定的协调操作，下面就是 DiagnosisEventSpout 类中使用的协调操作：

```
public class DefaultCoordinator implements BatchCoordinator<Long>,
                                           Serializable {
   private static final long serialVersionUID = 1L;
private static final Logger LOG =
            LoggerFactory.getLogger(DefaultCoordinator.class);

@Override
public boolean isReady(long txid) {
   return true;
}

@Override
public void close() {
}

@Override
public Long initializeTransaction(long txid,
                                  Long prevMetadata) {
   LOG.info("Initializing Transaction [" + txid + "]");
   return null;
   }
```

```
@Override
public void success(long txid) {
   LOG.info("Successful Transaction [" + txid + "]");
}
}
```

Trident spout 的第二个组成部分是 Emitter。在 Storm 里，spout 使用 collector 来发送 tuple，Emmiter 函数在 Trident spout 中执行这种功能。唯一的区别是，使用 TridentCollector 类，发送出去的 tuple 是通过 BatchCoordinator 类初始化的一批数据。

Emitter 方法的接口格式如下所示：

```
public interface Emitter<X> {
void emitBatch(TransactionAttempt tx, X coordinatorMeta,
               TridentCollector collector);
void close();
}
```

如前面代码所示，Emitter 函数只有一个功能，将 tuple 打包发射出去。为了实现这个功能，函数接收的参数包括 batch（由 coordinator 生成的）的元数据、事务的信息和 Emitter 用来发送 tuple 的 collector。DiagnosisEventEmitter 类的代码如下所示：

```
public class DiagnosisEventEmitter implements Emitter<Long>,
Serializable {

private static final long serialVersionUID = 1L;
AtomicInteger successfulTransactions = new AtomicInteger(0);

@Override
public void emitBatch(TransactionAttempt tx, Long
              coordinatorMeta, TridentCollector collector) {
  for (int i = 0; i < 10000; i++) {
      List<Object> events = new ArrayList<Object>();
      double lat =
            new Double(-30 + (int) (Math.random() * 75));
      double lng =
            new Double(-120 + (int) (Math.random() * 70));
      long time = System.currentTimeMillis();
      String diag = new Integer(320 +
                  (int) (Math.random() * 7)).toString();
      DiagnosisEvent event =
                  new DiagnosisEvent(lat, lng, time, diag);
      events.add(event);
      collector.emit(events);
  }
}
```

```
@Override
public void success(TransactionAttempt tx) {
   successfulTransactions.incrementAndGet();
}

@Override
public void close() {
}
}
```

发送的工作在 emitBatch() 中进行。例子中，我们随机分配一个经度和纬度，大体保持在美国范围内，使用 System.currentTimeMillis() 方法生成诊断的时间戳。

实际场景中，ICD-9-CM 的范围在 000 到 999 之间。针对本示例，我们仅使用 320 到 327 之间的诊断代码。这些代码如下所示：

代　码	描　述
320	细菌性脑膜炎
321	其他病原体所致脑膜炎
322	未知病因的脑膜炎
323	脑炎脊髓炎
324	颅内和脊椎内脓肿
325	静脉炎及颅内静脉窦血栓性静脉炎
326	颅内脓肿或化脓性感染后期影响
327	器官性睡眠障碍

这些诊断代码随机分配给事件。

在这个例子里，我们使用对象来封装诊断事件。为简化起见，我们将事件的每个组成部分作为 tuple 的一个独立字段。这里，对象封装还是使用 tuple 字段进行封装，需要权衡。通常会限制 tuple 的字段在易于管理的数量之内，但为了数据流控制或 tuple 的分组策略，将数据放在 tuple 的字段里还是有意义的。

在我们的例子中，DiagnosisEvent 类表示 topology 处理的关键数据。对象的代码如下所示：

```
public class DiagnosisEvent implements Serializable {
    private static final long serialVersionUID = 1L;
    public double lat;
    public double lng;
    public long time;
    public String diagnosisCode;

    public DiagnosisEvent(double lat, double lng,
                          long time, String diagnosisCode) {
```

```
        super();
        this.time = time;
        this.lat = lat;
        this.lng = lng;
        this.diagnosisCode = diagnosisCode;
        }
}
```

这个对象是一个简单的 JavaBean。时间戳使用 long 变量存储，存储的是纪元时间的秒数。经度和纬度使用 dobule 存储。diagnosisCode 类使用 string，以防系统可能需要处理非 ICD-9 数据，比如有字母的代码。

至此，topology 已经可以发射事件了。在实际场景中，我们可能将 topology 集成到一个医疗请求处理系统或者一个电子健康记录系统来进行实践演练。

3.4　Trident 运算

时间戳已经生成好了，下一步是加入处理事件的逻辑组件。在 Trident 中，这些组件称为运算（operation）。在我们的 topology 中，使用两种不同的运算：filter 和 function。

运算通过 Stream 对象的方法来调用。这个例子中，我们使用了 Stream 对象的下述方法：

```
public class Stream implements IAggregatableStream {
public Stream each(Fields inputFields, Filter filter) {
...
}

public IAggregatableStream each(Fields inputFields,
Function function,
Fields functionFields){
   ...
}

public GroupedStream groupBy(Fields fields) {
   ...
   }

public TridentState persistentAggregate(
StateFactory stateFactory,
CombinerAggregator agg,
Fields functionFields) {
   ...
}
}
```

注意前面代码中列出的方法返回形式为 Stream 对象或者 TridentState 对象，返回可以用来创建新的数据流。因此，运算可以连在一起使用流式接口形式的 Java 代码。让我们再看看示例 topology 中的关键代码：

```
inputStream.each(new Fields("event"), new DiseaseFilter())
   .each(new Fields("event"), new CityAssignment(),
         new Fields("city"))

   .each(new Fields("event", "city"),
         new HourAssignment(),
       new Fields("hour", "cityDiseaseHour"))

   .groupBy(new Fields("cityDiseaseHour"))

   .persistentAggregate(new OutbreakTrendFactory(),
        new Count(), new Fields("count")).newValuesStream()

   .each(new Fields("cityDiseaseHour", "count"),
         new OutbreakDetector(), new Fields("alert"))

   .each(new Fields("alert"), new DispatchAlert(),
         new Fields());
```

通常，应用运算需要声明一个输入域集合和一个输出域集合，也就是 funcition 域。上面代码中 topology 第二行声明我们需要 CityAssignment 对数据流中的每个 tuple 执行操作。在每个 tuple 中，CityAssignment 会在 event 字段上运算并且增加一个叫做 city 的新字段，这个字段会附在 tuple 中向后发射。

每个操作在流式风格的语法上略有不同，这取决于操作需要哪些信息。下面将介绍不同操作的详细语法和语义。

3.4.1 Trident filter

我们 topology 逻辑中的第一部分就是个过滤器 filter，它会忽略掉我们不关心的疾病事件。在这个例子中，系统只关心脑膜炎（meningitis）的病情，从前面表格中看到，脑膜炎对应的疾病代码是 320、321 和 322。

为了通过疾病代码过滤事件，我们需要利用 Trident filter。Trident 通过提供 BaseFilter 类，我们通过实现子类就可以方便地对 tuple 进行过滤，滤除系统不需要的 tuple。BaseFilter 类实现了 Filter 接口，这个接口如下面代码片段所示：

```
public interface Filter extends EachOperation {
    boolean isKeep(TridentTuple tuple);
}
```

为了在数据流中过滤 tuple，应用需要通过继承 BaseFilter 类来实现这个接口。这个例子中，我们使用下述过滤器过滤事件：

```
public class DiseaseFilter extends BaseFilter {
private static final long serialVersionUID = 1L;
private static final Logger LOG =
LoggerFactory.getLogger(DiseaseFilter.class);

@Override
public boolean isKeep(TridentTuple tuple) {
   DiagnosisEvent diagnosis = (DiagnosisEvent) tuple.getValue(0);
   Integer code = Integer.parseInt(diagnosis.diagnosisCode);
   if (code.intValue() <= 322) {
       LOG.debug("Emitting disease [" +
diagnosis.diagnosisCode + "]");
       return true;
   } else {
       LOG.debug("Filtering disease [" +
diagnosis.diagnosisCode + "]");
       return false;
   }
}
}
```

上面的代码中，我们从 tuple 中提取了 DiagnosisEvent 并且检查疾病代码。因为所有的脑膜炎代码小于等于 322，我们也没有发送其他代码，所以只需要简单地检查代码是否小于 322，就可以决定事件是否和脑膜炎有关。

Filter 操作结果返回 True 的 tuple 将会被发送到下游进行操作。如果方法返回 False，该 tuple 就不会发送到下游。

在我们的 topology 中，我们在数据流上使用 each（inputFields，filter）方法，将这个过滤器应用到数据流的每个 tuple 中：

```
inputStream.each(new Fields("event"), new DiseaseFilter())
```

3.4.2　Trident function

在 filter 之外，Storm 还提供了一个更通用功能的接口 function。function 和 Storm 的 bolt 类似，读取 tuple 并且发送新的 tuple。其中一个区别是，Trident function 只能添加数

据。function 发送数据时，将新字段添加在 tuple 中，并不会删除或者变更已有的字段。

function 接口如下代码片段所示：

```
public interface Function extends EachOperation {
void execute(TridentTuple tuple, TridentCollector collector);
}
```

和 Storm 的 bolt 类似，function 实现了一个包括实际逻辑的方法 execute。function 的实现也可以选用 TridentCollector 来发送 tuple 到新的 function 中。用这种方式，function 也可以用来过滤 tuple，起到 filter 的作用。

我们 topology 中的第一个 function 是 CityAssignment，如下所示：

```
public class CityAssignment extends BaseFunction {
private static final long serialVersionUID = 1L;
private static final Logger LOG = LoggerFactory.
getLogger(CityAssignment.class);

private static Map<String, double[]> CITIES =
                        new HashMap<String, double[]>();

    { // Initialize the cities we care about.
        double[] phl = { 39.875365, -75.249524 };
        CITIES.put("PHL", phl);
        double[] nyc = { 40.71448, -74.00598 };
        CITIES.put("NYC", nyc);
        double[] sf = { -31.4250142, -62.0841809    };
        CITIES.put("SF", sf);
        double[] la = { -34.05374, -118.24307 };
        CITIES.put("LA", la);
    }

    @Override
    public void execute(TridentTuple tuple,
TridentCollector collector) {
        DiagnosisEvent diagnosis =
                            (DiagnosisEvent) tuple.getValue(0);
        double leastDistance = Double.MAX_VALUE;
        String closestCity = "NONE";

        // Find the closest city.
        for (Entry<String, double[]> city : CITIES.entrySet()) {
            double R = 6371; // km
            double x = (city.getValue()[0] - diagnosis.lng) *
                    Math.cos((city.getValue()[0] + diagnosis.lng) / 2);
            double y = (city.getValue()[1] - diagnosis.lat);
            double d = Math.sqrt(x * x + y * y) * R;
```

```
            if (d < leastDistance) {
                leastDistance = d;
                closestCity = city.getKey();
            }
        }

        // Emit the value.
        List<Object> values = new ArrayList<Object>();
        values.add(closestCity);
        LOG.debug("Closest city to lat=[" + diagnosis.lat +
                  "], lng=[" + diagnosis.lng + "] == ["
                  + closestCity + "], d=[" + leastDistance + "]");
        collector.emit(values);
    }
}
```

在这个 function 中，我们使用静态初始化的方式建立了一个我们关心的城市的地图。示例中，function 包括一个地图，存储了的坐标信息包括：Philadelphia（PHL）、New York City（NYC）、San Francisco（SF）和 LosAngeles（LA）。

在 execute() 方法中，函数遍历城市计算事件和城市之间的距离。现实系统中，地理空间的索引效率会高很多。

function 声明的字段数量必须和它发射出值的字段数一致。如果不一致，Storm 就会抛出 IndexOutOfBoundsException 异常。

我们 topology 中的下一个 function 是 HourAssignment，用来转化 Unix 时间戳为纪元时间的小时，可以用来对事件发生进行时间上的分组操作。HourAssignment 的代码如下：

```
public class HourAssignment extends BaseFunction {
private static final long serialVersionUID = 1L;
private static final Logger LOG =
            LoggerFactory.getLogger(HourAssignment.class);

@Override
public void execute(TridentTuple tuple,
                    TridentCollector collector) {
   DiagnosisEvent diagnosis = (DiagnosisEvent) tuple.getValue(0);
   String city = (String) tuple.getValue(1);

   long timestamp = diagnosis.time;
   long hourSinceEpoch = timestamp / 1000 / 60 / 60;

   LOG.debug("Key =  [" + city + ":" + hourSinceEpoch + "]");
   String key = city + ":" + diagnosis.diagnosisCode + ":" +
```

```
                hourSinceEpoch;

    List<Object> values = new ArrayList<Object>();
    values.add(hourSinceEpoch);
    values.add(key);
    collector.emit(values);
}
}
```

我们重写了这个 function，同时发射了小时的数值，以及由城市、疾病代码、小时组合而成的 key。实际上，这个组合值会作为聚合计数的唯一标识符，后面会详细解释。

我们 topology 中最后两个 funciton 是用来侦测疾病暴发并且告警的。OutbreakDetector 类的代码如下：

```
public class OutbreakDetector extends BaseFunction {
    private static final long serialVersionUID = 1L;
    public static final int THRESHOLD = 10000;

    @Override
    public void execute(TridentTuple tuple,
                        TridentCollector collector) {
  String key = (String) tuple.getValue(0);
  Long count = (Long) tuple.getValue(1);

  if (count > THRESHOLD) {
      List<Object> values = new ArrayList<Object>();
      values.add("Outbreak detected for [" + key + "]!");
      collector.emit(values);
  }
}
}
```

这个 function 提取出了特定城市、疾病、时间的发生次数，并且检查计数是否超过了设定的阈值。如果超过，发送一个新的字段包括一条告警信息。在上面代码里，注意这个 function 实际上扮演了一个过滤器的角色，但是却作为一个 function 的形式来实现，是因为需要在 tuple 中添加新的字段。因为 filter 不能改变 tuple，当我们既想过滤又想添加字段时必须使用 function。

最后一个 function 的功能就是发布一个告警（并且结束程序）。代码如下：

```
public class DispatchAlert extends BaseFunction {
    private static final long serialVersionUID = 1L;

    @Override
```

```
    public void execute(TridentTuple tuple,
                        TridentCollector collector) {
    String alert = (String) tuple.getValue(0);
    Log.error("ALERT RECEIVED [" + alert + "]");
    Log.error("Dispatch the national guard!");
    System.exit(0);
    }
}
```

这个方法非常简单，提取了告警的内容，并写入日志，最后结束整个程序。

3.5　Trident 聚合器

和 function 类似，aggregator（聚合器）允许 topology 组合 tuple。不同的是，它会替换 tuple 的字段和值。有三种聚合器：CombinerAggregator、ReducerAggregator 和 Aggregator。

3.5.1　CombinerAggregator

CombinerAggregator 用来将一个集合的 tuple 组合到一个单独的字段中，Combiner 的签名（Signature）如下所示：

```
public interface CombinerAggregator {
   T init (TridentTuple tuple);
   T combine(T val1, T val2);
   T zero();
}
```

Storm 对每个 tuple 调用 init() 方法，然后重复调用 combine() 方法直到一个分片的数据处理完成。传递给 combine() 方法的两个参数是局部聚合的结果，以及调用了 init() 返回的值。分片会在后面的部分详细介绍，分片实际上就是 tuples 组成的数据流在同一个机器上的一个子集。将 tuple 生成的值进行组合之后，Storm 发送组合结果作为一个新的字段。如果分片是空的，Storm 会发送 zero() 方法执行的返回。

3.5.2　ReducerAggregator

ReducerAggregator 接口有一点区别，签名如下：

```
public interface ReducerAggregator<T> extends Serializable {
    T init();
    T reduce(T curr, TridentTuple tuple);
}
```

Storm 调用 init() 方法来获取原始值。然后为分片中的每一个 tuple 调用 reduce() 方法，直到分片数据处理完成。第一个参数是局部的聚合结果。这个方法的实现需要将第二个参数 tuple 合并到局部聚合结果中返回。

3.5.3 Aggregator

最通用的聚合操作是 Aggregator。签名如下所示：

```
public interface Aggregator<T> extends Operation {
    T init(Object batchId, TridentCollector collector);
    void aggregate(T val, TridentTuple tuple,
TridentCollector collector);
 void complete(T val, TridentCollector collector);
}
```

Aggregator 接口的 aggregate() 方法和 function 接口的 execute() 方法类似，但是多了一个 value 参数。这样 Aggregator 就可以在处理 tuple 的时候累积值。注意，在 Aggregator 接口中，aggregate() 和 complete() 方法都有 collector 这个参数，通过它可以发射任意个数的 tuple。在我们的 topology 例子中，我们利用了一个内置的 Count 的 Aggregator。Count 的实现如下面代码片段所示：

```
public class Count implements CombinerAggregator<Long> {
    @Override
    public Long init(TridentTuple tuple) {
        return 1L;
    }

    @Override
    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }

    @Override
    public Long zero() {
        return 0L;
    }
}
```

我们在示例 topology 中使用了分组和计数来统计在一个城市附近一个小时内发生疾病的次数。实现代码如下所示：

```
.groupBy(new Fields("cityDiseaseHour"))
.persistentAggregate(new OutbreakTrendFactory(),
   new Count(), new Fields("count")).newValuesStream()
```

回顾 Storm 在不同机器上的数据的分片，如图 3-2 所示。

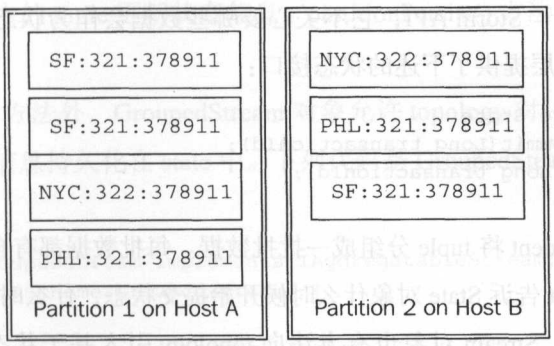


图　3-2

groupBy() 方法强制数据重新分片，将特定字段值相同的 tuple 分组到同一个分片中。为了做到这个，Storm 必须将相似的 tuple 发送到相同的主机上。图 3-3 展示了数据被 groupBy() 重新分组后的分片情况。

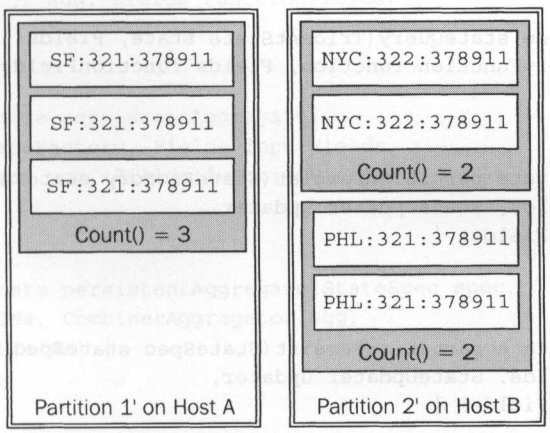


图　3-3

分片后，agreagte 函数在每个分片数据的每个分组中运行。在我们的例子里，根据城市、小时、疾病代码作为分组的关键词。然后 Count aggregator 在每个分组上执行，将计数发射给下游的消费者组件。

3.6　Trident 状态

我们现在已经给每个 aggregator 的分组数据进行了计数，现在想将信息进行持久化存

储，以便进一步分析。在 Trident 中，持久化操作从状态管理开始。Trident 对状态有底层的操作原语，但不同于 Storm API，它不关心要哪些数据会作为状态存储或者如何存储这些状态。Trident 在高层提供了下述的状态接口：

```
public interface State {
   void beginCommit(Long transactionId);
   void commit(Long transactionId);
}
```

上面提到了，Trident 将 tuple 分组成一批批数据。每批数据都有自己的事务标识符。在前面的接口中，Trident 告诉 State 对象什么时候开始提交状态，什么时候提交状态应该结束。

和 function 类似，Stream 对象也有方法向 topology 引入基于状态的操作。更具体说，Trident 有两种数据流：Stream 和 GroupedStream。一个 GroupedStream 是 GroupBy 操作的结果。在我们的 topology 中，我们根据 HourAssignment function 生成的 key 对 tuple 进行分组。

在 Steam 对象中，下列方法允许 topology 读和写状态信息：

```
public class Stream implements IAggregatableStream {
    ...
    public Stream stateQuery(TridentState state, Fields inputFields,
           QueryFunction function, Fields functionFields) {
    ...
 }

public TridentState partitionPersist(StateFactory stateFactory,
Fields inputFields, StateUpdater updater,
Fields functionFields) {
   ...
}

public TridentState partitionPersist(StateSpec stateSpec,
Fields inputFields, StateUpdater updater,
Fields functionFields) {
   ...
}

public TridentState partitionPersist(StateFactory stateFactory,
Fields inputFields, StateUpdater updater) {
   ...
   }

public TridentState partitionPersist(StateSpec stateSpec,
Fields inputFields, StateUpdater updater) {
    ...
}
...
}
```

stateQuery() 方法从 state 生成了一个输入流，不同参数的几个 partitionPersist() 方法允许 topology 从数据流中的 tuple 更新状态信息。partitionPersist() 方法的操作对象是每个数据分片。

在 Stream 对象的方法外，GroupedStream 对象允许 topology 对一批 tuple 进行聚合统计，并且将收集到的信息持久化在 state 中。下列代码是 GroupedSteam 类中和状态相关的方法：

```
public class GroupedStream implements IAggregatableStream,
GlobalAggregationScheme<GroupedStream> {
...
   public TridentState persistentAggregate(
StateFactory stateFactory, CombinerAggregator agg,
Fields functionFields) {
...
}

public TridentState persistentAggregate(StateSpec spec,
CombinerAggregator agg, Fields functionFields) {
...
}

public TridentState persistentAggregate(
StateFactory stateFactory, Fields inputFields,
CombinerAggregator agg, Fields functionFields) {
...
}

public TridentState persistentAggregate(StateSpec spec,
Fields inputFields, CombinerAggregator agg,
Fields functionFields) {
...
}

public TridentState persistentAggregate(
StateFactory stateFactory, Fields inputFields,
ReducerAggregator agg, Fields functionFields) {
...
}

public TridentState persistentAggregate(StateSpec spec, Fields
inputFields, ReducerAggregator agg, Fields functionFields) {
...
}

public Stream stateQuery(TridentState state, Fields inputFields,
```

```
QueryFunction function, Fields functionFields) {
...
}

public TridentState persistentAggregate(
StateFactory stateFactory, ReducerAggregator agg,
Fields functionFields) {
...
}

public TridentState persistentAggregate(StateSpec spec,
ReducerAggregator agg, Fields functionFields) {
...
}

public Stream stateQuery(TridentState state,
    QueryFunction function, Fields functionFields) {
...
}
}
```

和 Steam 对象类似，stateQuery() 方法从 State 生成一个输入数据流。不同参数的几个 persistAggregate() 方法允许 topology 从数据流中的 tuple 更新状态信息。注意 GroupedStream 方法有一个 Aggregator 参数，它在信息写入 State 对象之前执行。

现在考虑将这些 function 应用到我们的例子中来。在我们的系统中，需要将事件发生的城市、疾病代码、每小时内产生疾病统计量进行持久存储。这样可以生成报表如表 3-2 所示。

表 3-2

疾　病	城　市	日　期	时　间	疾病统计量
细菌性脑膜炎	San Francisco	3/12/2013	3:00 PM	12
细菌性脑膜炎	San Francisco	3/12/2013	4:00 PM	50
细菌性脑膜炎	San Francisco	3/12/2013	5:00 PM	100
天花	New York	3/13/2013	5:00 PM	6

为了实现这个功能，我们需要将聚集操作中生成的统计量进行持久化存储。我们可以使用 groupBy 函数返回的 GroupedStream 接口（如前面所示）调用 persistAggregate 方法。下面代码是示例 topology 中具体的调用方式：

```
persistentAggregate(new OutbreakTrendFactory(),
  new Count(), new Fields("count")).newValuesStream()
```

要了解持久化存储，我们首先来看这个方法的第一个参数。Trident 使用一个工厂类

来生成 State 的实例。OutbreakTrendFactory 是我们的 topology 提供给 Storm 的工厂类。OutbreakTrendFactory 代码如下：

```
public class OutbreakTrendFactory implements StateFactory {
private static final long serialVersionUID = 1L;

@Override
public State makeState(Map conf, IMetricsContext metrics,
int partitionIndex, int numPartitions) {
   return new OutbreakTrendState(new OutbreakTrendBackingMap());
}
}
```

工厂类返回一个 State 对象，Storm 用它来持久化存储信息。在 Storm 中，有三种类型的状态。每个类型的描述如表 3-3 所示。

表　3-3

状态类型	说　明
非事务型	没有回滚能力，更新操作是永久性的，commit 操作会被忽略
重复事务型	由同一批 tuple 提供的结果是幂等的
不透明事务型	更新操作基于先前的值，这样一批数据组成不同，持久化的数据也会变

在分布式环境下，数据可能被重放，为了支持计数和状态更新，Trident 将状态更新操作进行序列化，使用不同的状态更新模式对重放和错误数据进行容错。接下来会介绍这些模式。

3.6.1　重复事务型状态

在重复事务型状态中，最后一批提交的数据的标识符存在数据中。当且仅当一批数据标识符的序号大于当前标识符时，才进行更新操作。如果小于或者等于当前标识符，将会忽略更新操作。

为了演示这个实现方法，考虑如表 3-4 所示的数据批次的序列，这些记录对我们例子中的数据按照 key 进行聚合计数。

表　3-4

批　次	状态更新
1	{SF:320:378911 = 4}
2	{SF:320:378911 = 10}
3	{SF:320:378911 = 8}

这些批次数据按照下列将顺序处理完成：

1 à 2 à 3 à 3（重放）

处理结果将按照表 3-5 中的状态变更操作，中间的一列数据用来存储数据标识符，记录最近一次合并进状态的数据批次编号。

表 3-5

处理过的批次	状 态	
1	{ Batch = 1 }	{ SF:320:378911 = 4 }
2	{ Batch = 2 }	{ SF:320:378911 = 14 }
3	{ Batch = 3 }	{ SF:320:378911 = 22 }
3（重放）	{ Batch = 3 }	{ SF:320:378911 =22}

注意当批次号为 3 的数据完成重放后，不会对状态产生影响，因为 Trident 已经将其更新合并到 State 中。为了让重复事务型状态正常工作，一批数据在重放时的数据组成不能变化。

3.6.2 不透明型状态

重复事务型状态的实现依赖于数据批次中包含数据保持不变，这种特性在系统遇到错误时就可能保证不了了。如果发射数据的 spout 发生了局部故障，原始批次数据中的部分 tuple 可能无法重新发送。不透明型状态通过存储当前的状态和前一次状态来允许批次的数据组成发生变化。

假设我们的数据批次和前面的例子相同，这次在批次号为 3 的数据重新发送时，聚合统计量会不一样，因为包括了不同的 tuple 的集合，如表 3-6 所示。

表 3-6

批 次	状态更新
1	{SF:320:378911 = 4}
2	{SF:320:378911 = 10}
3	{SF:320:378911 = 8}
3（重放）	{SF:320:378911 = 6}

使用不透明型状态时，状态信息的更新如表 3-7 所示。

表 3-7

处理过的批次	提交过的批次	前一个状态	当前状态
1	1	{ }	{ SF:320:378911 = 4 }
2	2	{ SF:320:378911 = 4 }	{ SF:320:378911 = 14 }
3（应用）	3	{ SF:320:378911 = 14 }	{ SF:320:378911 = 22 }
3（重放）	3	{ SF:320:378911 = 14 }	{ SF:320:378911 = 20 }

注意不透明型状态存储了上一个状态信息；因此，当第 3 批次数据重放时，可以使用新的聚合计数重新赋值。

你可能会好奇，为什么可以在一批数据提交后还会再次应用这批数据。对应的一种场景是状态已经更新成功了，但是下游处理失败。在我们的例子中，可能是告警信息发布失败。这种情况下 Trident 会重新发送这批数据。在最坏的情况下，当要求 spout 重新发送这批数据时，可能有一个或者多个数据源不可用。

在事务型 spout 中，需要一直等待直到数据源恢复可用。不透明事务型 spout 会发送当前可用的数据分片，数据的处理照常进行。因为 Trident 是按照序列处理数据批次并记录状态，因此每个单独的批次都不能延迟，因为延迟可能导致阻塞整个系统。

在实现中，状态类型的选择需要基于 spout 来保证处理行为的幂等性，才能保证不会错误计数或者破坏了状态。表 3-8 展示了为保证幂等性的可行搭配。

表　3-8

spout 类型	非事务型状态	不透明型状态	重复事务型状态
非事务型 spout			
不透明型 spout		×	
事务型 spout		×	×

幸运的是，Storm 提供了 map 的实现来屏蔽了持久层进行状态管理的复杂性。尤其是，Trident 提供了 State 实现通过维护额外的信息来实现上面提到的保证。这些对象命名相似：NonTransactionalMap、TransactionalMap、OpaqueMap。

回到我们的例子里，因为我们没有事务型保证，所以选用 NonTransactionalMap 作为我们的 State 对象。

OutbreakTrendState 对象如下代码所示：

```
public class OutbreakTrendState extends NonTransactionalMap<Long> {
protected OutbreakTrendState(
OutbreakTrendBackingMap outbreakBackingMap) {
   super(outbreakBackingMap);
}
}
```

上面代码利用了 MapState 对象，只需要传递一个 backing map。我们的例子中，使用 OutbreakTrendBackingMap。代码如下：

```
public class OutbreakTrendBackingMap implements IBackingMap<Long> {
    private static final Logger LOG =
LoggerFactory.getLogger(OutbreakTrendBackingMap.class);
```

```
 Map<String, Long> storage =
new ConcurrentHashMap<String, Long>();

 @Override
 public List<Long> multiGet(List<List<Object>> keys) {
    List<Long> values = new ArrayList<Long>();
    for (List<Object> key : keys) {
        Long value = storage.get(key.get(0));
        if (value==null){
            values.add(new Long(0));
        } else {
            values.add(value);
        }
    }
    return values;
}

@Override
public void multiPut(List<List<Object>> keys, List<Long> vals) {
    for (int i=0; i < keys.size(); i++) {
        LOG.info("Persisting [" + keys.get(i).get(0) + "] ==> ["
+ vals.get(i) + "]");
        storage.put((String) keys.get(i).get(0), vals.get(i));
    }
}
}
```

在我们的 topology 中，实际上没有固化存储数据。我们简单地将数据放入 Concurrent HashMap 中。显然，对于多个机器的环境下，这样是不可行的。然而 BackingMap 是一个非常巧妙的抽象。只需要将传入 MapState 对象的 backing map 的实例替换就可以更换持久层的实现。我们在后面章节会看到例子。

3.7 执行 topology

OutbreakDetectionTopology 类有下列方法：

```
public static void main(String[] args) throws Exception {
    Config conf = new Config();
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("cdc", conf, buildTopology());
    Thread.sleep(200000);
    cluster.shutdown();
}
```

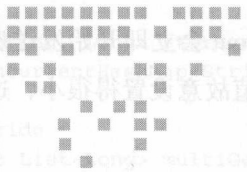
执行这个方法会将 topology 部署到本地集群中。spout 会立即开始发送疾病事件，由 Count aggregator 收集计数。OutbreakDetector 类中的阈值故意设置得很小，这样计数很快就超过阈值，这时程序结束，输出如下日志：

```
INFO [Thread-18] DefaultCoordinator.success(31) | Successful Transaction
[8]
INFO [Thread-18] DefaultCoordinator.initializeTransaction(25) |
Initializing Transaction [9]
...
INFO [Thread-24] OutbreakTrendBackingMap.multiPut(34) | Persisting
[SF:320:378951] ==> [10306]
INFO [Thread-24] OutbreakTrendBackingMap.multiPut(34) | Persisting
[PHL:320:378951] ==> [893]
INFO [Thread-24] OutbreakTrendBackingMap.multiPut(34) | Persisting
[NYC:322:378951] ==> [1639]
INFO [Thread-24] OutbreakTrendBackingMap.multiPut(34) | Persisting
[SF:322:378951] ==> [10254]
INFO [Thread-24] OutbreakTrendBackingMap.multiPut(34) | Persisting
[SF:321:378951] ==> [10386]
...
00:04 ERROR: ALERT RECEIVED [Outbreak detected for [SF:320:378951]!]
00:04 ERROR: Dispatch the National Guard!
```

注意当数据批次成功处理完成时会通知到 coordinator，并且在几批数据后，就超过阈值了，系统会通过错误消息告知我们：派遣国民警卫队！

总结

在本章中，我们建立了一个 topology 处理疾病信息来监测异常情况，这些异常可能说明有疾病暴发。这个数据流也可以应用到任何类型的数据上，包括天气信息、地震信息或者交通信息。我们运用 Trident 中的基本原语来构建一个系统，对事件进行统计，即使数据重放也适用。在本书后面的章节中，我们会利用同样的构造和模式来完成类似的功能。



Chapter 4

第 4 章 实时趋势分析

在本章中，我们将介绍使用 Storm 和 Trident 实现的趋势分析计数。实时趋势分析涉及从数据流中识别模式，比如特定事件的发生概率或者计数达到了特定的阈值。常见的例子包括发现社交媒体中话题的趋势，比如在 Twitter 上一个特定话题的流行，搜索引擎中一个词汇被搜索的趋势。Storm 起源于一个对 Twitter 数据进行实时分析的项目，并且提供了分析计算所需要的多个关键功能。

在前面的章节中，主要用来模拟数据生成的 spout 使用静态或者随机生成的数据。在本章中，我们会引入一个开源的 spout 实现，可以从一个队列（Apache Kafka）里取数据来发射并且支持三种类型的 Trident spout 事务（非事务型、重复事务型和不透明事务型）。我们将会用流行的日志框架 logback 实现一个简单的、通用的方法将数据填入 Kafka 队列，这样现有程序稍加改动就可以迅速使用实时分析系统了。

在本章中，我们将包括以下主题：

- 将数据作为日志写入 Apache Kafka 队列并且作为数据流发送给 Storm。
- 将现有应用程序的日志发送给 Storm 作分析。
- 实现一个指数加权移动平均数的 Trident function。
- Storm 使用 XMPP 协议发送告警和通知。

4.1　应用场景

在例子中，我们有一个或者一组应用程序（网站、企业应用，等等）都使用流行的日志框架 logback（http://logback.qos.ch/）将结构化的日志消息写入磁盘（访问记录、错误信息，等等）。当前，在这些数据上实时分析的唯一方法是处理文件并且在类似 Hadoop 这样的系统上进行批处理操作。处理的延迟明显减慢了我们的反应时间，从 log 数据中找出模式的时间往往以小时或者天来计，一个特定事件发生后，可能已经来不及响应了。如果在模式出现时能积极感知，而不是事后才发现，这是非常令人期待的事情。

这个使用场景代表了一个通用主题，在众多商业场景中，都有广泛的应用空间，包括下述应用：

- 应用监控：例如当网络中的错误到达一定比率后，通知系统管理员。
- 入侵检测：例如检测异常行为，登录尝试失败次数增加。
- 供应链管理：例如用来检测特定商品销售峰值并且实时调整订货。
- 在线推荐：例如发现流行趋势并且动态调整推送广告。

4.2　体系结构

我们程序的体系结构如图 4-1 所示，包含多个组件。

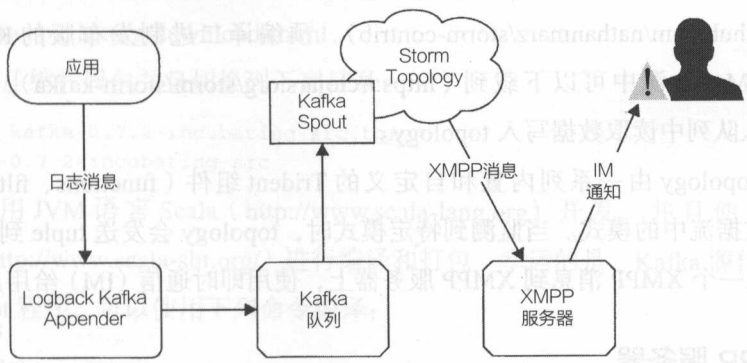


图　4-1

4.2.1　数据源应用程序

数据源应用程序指使用了 logback 框架来记录 log 信息的任意程序。为实现我们的目的，

我们开发了一个简单的程序，按照特定的时间间隔发送结构化的日志消息。然而，像上面介绍过的，任何使用了 logback 或者 slf4j 框架的应用程序都可以通过配置替代这个程序。

4.2.2 logback Kafka appender

Logback 框架的扩展机制允许在配置里自定义额外的输出源（appender）。logback appender 是一个 Java 类，接收日志事件并处理。最通用的 appender 是 FileAppender 子类中的一个，简单的格式化日志并且写入磁盘中的文件。其他 appender 实现会将数据写到网络 socket 中、关系数据库中，或作为邮件通知发送到 SMTP 服务上。我们会实现一个 appender 将日志写到 Apache Kafka 队列中。

4.2.3 Apache Kafka

Apache Kafka（http://kafka.apache.org/）是一个开源分布式“发布 – 订阅”消息系统。Kafka 为高吞吐、持久化实时数据流进行了特别设计和优化。和 Storm 类似，Kafka 设计足以支撑商业软件的大规模水平扩展，支持每秒数十万消息的处理。

4.2.4 Kafka spout

Kafka spout 从 Kafka 队列中读取数据，并且发射到 Storm 或者 Trident topology 中。Kafka spout 最初由 Nathan Marz 开发，现在仍然是 GitHub 上 storm-contrib 项目中的一部分（https://github.com/nathanmarz/storm-contrib）。预编译二进制发布版的 Kafka spout 在 clojars.org 的 Maven 源中可以下载到（https://clojars.org/storm/storm-kafka）。我们将使用 Kafka spout 从队列中读取数据写入 topology。

我们的 topology 由一系列内置和自定义的 Trident 组件（functions、filters、state 等）构成，监测数据流中的模式。当监测到特定模式时，topology 会发送 tuple 到一个 function 里，它会发送一个 XMPP 消息到 XMPP 服务器上，使用即时通信（IM）给用户发送通知。

4.2.5 XMPP 服务器

可扩展通信和表示协议（Extensible Messaging and Presence Protocol，XMPP）（http://xmpp.org）是一种基于 XML 的标准，用来进行及时通信、展示信息和通信录维护。很多 IM 客户端比如 Adium（OS X 下 http://adium.im/）、Pidgin（OS X、Linux 和 Windows 下

http://www.pidgin.im/）都支持 XMPP 协议，如果你曾经用过 Google Talk 作为即时通信工具，那么你已经接触过 XMPP 了。

为了简化 OS X、Linux 和 Windows 之间的兼容性，我们将使用开源 OpenFire XMPP 服务。

4.3　安装需要的软件

我们首先来安装必需的软件：Apache Kafka 和 OpenFire。虽然 Kafka 是一个分布式消息系统，在单一节点上，甚至本地开发环境下也可以正常使用。在生产环境中，你需要根据需求规模建立一个单机或者多机器集群。OpenFire 服务不是集群系统，可以直接安装在单节点或者本地。

4.3.1　安装 Kafka

和 Storm 类似，Kafuka 队列依赖 ZooKeeper 来存储状态信息。因为 Storm 使用 ZooKeeper 的负载较轻，多数情况下是可以直接和 Kafka 队列共享一个 ZooKeeper 实例。因为我们已经在第 2 章中讲了如何安装 ZooKeeper，这里就只讲如何在开发环境中为 Kafka 运行 ZooKeeper。

首先从下列网址官网下载 0.7.x 版本的 Apache Kafka 发行版：

http://kafka.apache.org/downloads.html

其次，解压缩代码包并且切换到下列目录：

```
tar -zxf kafka-0.7.2-incubating-src.tgz
cd kafka-0.7.2-incubating-src
```

Kafka 是用 JVM 语言 Scala（http://www.scala-lang.org）开发，并且使用 sbt（Scala Build Tool）（http://www.scala-sbt.org/）进行编译和打包。幸运的是，Kafka 源代码发行版中已经包括了 sbt 程序，可以使用下列命令编译：

```
./sbt update package
```

在 Kafka 启动之前，除非已经有一个运行中的 ZooKeeper 服务，否则需要先运行 Kafaka 中打包的 ZooKeeper 服务。执行下列命令：

```
./bin/zookeeper-server-start.sh ./config/zookeeper.properties
```

最后，在终端中，启动 Kafka 服务：

```
./bin/kafka-server-start.sh ./config/server.properties
```

Kafka 服务现在就可以使用了。

4.3.2 安装 OpenFire

OpenFire 在 OS X 和 Windows 里都有安装文件，Linux 里不同发行版本也都有软件包，可以从下面的网站下载：

```
http://www.igniterealtime.org/downloads/index.jsp
```

为了安装 OpenFire，下载操作系统对应的安装包，按照正确的安装指令进行安装。安装指南见下面的网站：

```
http://www.igniterealtime.org/builds/openfire/docs/latest/
documentation/index.html
```

4.4 示例程序

程序组件是一个简单的 Java 类，使用了 Logging Facade for Java（SLF4J）（http://www.slf4j.org）来记录日志。我们会模拟一个程序，按照很慢的频率生成告警信息，然后切换到更快的频率来生成告警信息，最后再减慢频率：

- 每 5 秒记录一条告警日志，持续 30 秒（慢速）
- 每 1 秒记录一条告警日志，持续 15 秒（快速）
- 每 5 秒记录一条告警日志，持续 30 秒（慢速）

这个程序的目的是生成一个简单的模式，当发送频率状态变更，特定的模式显现时，我们的 topology 可以识别这个模式并且通过发送通知来响应。代码片段如下：

```
public class RogueApplication {
    private static final Logger LOG = LoggerFactory.
getLogger(RogueApplication.class);

    public static void main(String[] args) throws Exception {
        int slowCount = 6;
        int fastCount = 15;
        // slow state
        for(int i = 0; i < slowCount; i++){
            LOG.warn("This is a warning (slow state).");
            Thread.sleep(5000);
```

```
        }
        // enter rapid state
        for(int i = 0; i < fastCount; i++){
            LOG.warn("This is a warning (rapid state).");
            Thread.sleep(1000);
        }
        // return to slow state
        for(int i = 0; i < slowCount; i++){
            LOG.warn("This is a warning (slow state).");
            Thread.sleep(5000);
        }
    }
}
```

发送日志消息到 Kafka

Logback 框架提供了简单的扩展机制，允许添加额外的 appender。在我们的例子中，我们会实现一个 appender，将日志消息数据写到 Kafka 中。

Logback 包括一个 ch.qos.logback.core.AppenderBase 抽象类，简化了对 Appender 接口的实现。AppenderBase 类定义一个如下的抽象方法：

```
abstract protected void append(E eventObject);
```

eventObject 参数表示一个日志事件，并且包括事件的属性，比如发生时间、日志级别（DEBUG、INFO、WARN，等等），以及事件消息本身。我们会重写 append 方法，AppenderBase 类定义了两个额外的声明周期的方法需要重写：

```
public void start();
public void stop();
```

start() 方法在 logback 框架初始化时调用，stop() 方法在对象释放时执行。我们重写这些方法建立和断开到 Kafka 服务的连接。

KafkaAppender 类的代码如下所示：

```
public class KafkaAppender extends AppenderBase<ILoggingEvent> {

    private String topic;
    private String zookeeperHost;
    private Producer<String, String> producer;
    private Formatter formatter;

    // java bean definitions used to inject
    // configuration values from logback.xml
    public String getTopic() {
```

```
        return topic;
    }

    public void setTopic(String topic) {
        this.topic = topic;
    }

    public String getZookeeperHost() {
        return zookeeperHost;
    }

    public void setZookeeperHost(String zookeeperHost) {
        this.zookeeperHost = zookeeperHost;
    }

    public Formatter getFormatter() {
        return formatter;
    }

    public void setFormatter(Formatter formatter) {
        this.formatter = formatter;
    }

    // overrides
    @Override
    public void start() {
        if (this.formatter == null) {
            this.formatter = new MessageFormatter();
        }
        super.start();
        Properties props = new Properties();
        props.put("zk.connect", this.zookeeperHost);
        props.put("serializer.class", "kafka.serializer.
StringEncoder");
        ProducerConfig config = new ProducerConfig(props);
        this.producer = new Producer<String, String>(config);
    }

    @Override
    public void stop() {
        super.stop();
        this.producer.close();
    }

    @Override
    protected void append(ILoggingEvent event) {
        String payload = this.formatter.format(event);
        ProducerData<String, String> data = new ProducerData<String,
```

```
String>(this.topic, payload);
            this.producer.send(data);
        }
    }
```

如你看到的，logback 框架初始化时，这个类中 JavaBean 风格的访问器允许我们通过依赖注入方式在运行时改变配置的值。zookeeperHosts 属性的 setter 和 getter 方法用来初始化 KafkaProducer 客户端，配置这个属性可以用来发现 ZooKeeper 中已经注册的 Kafka 主机。一个替代的方法是提供一个静态的 Kafka 机器列表，但是为简单起见就使用了自动发现的机制。topic 属性用来告诉 KafkaConsumer 客户端从哪个 Kafka 主题中读取数据。

Formatter 属性比较特殊。我们定义了这个接口，像下面代码这样提供了一个扩展的点用来处理结构化（可解析）的日志信息。

```
public interface Formatter {
    String format(ILoggingEvent event);
}
```

Formatter 实现的作用是接收一个 ILoggingEvent 对象并且返回一个机器可读的字符串供消费者处理。下面的代码是一个简单的实现，返回了日志信息，并且丢掉了额外的元数据：

```
public class MessageFormatter implements Formatter {

    public String format(ILoggingEvent event) {
        return event.getFormattedMessage();
    }
}
```

下面的 logback 配置文件示例了如何使用 appender。这个例子没有自定义 Formatter 实现，所以 KafkaAppender 类会默认使用 MessageFormatter 类，并且仅仅将日志消息写到 Kafka 队列中并且丢弃日志事件中额外的信息，代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <appender name="KAFKA"
        class="com.github.ptgoetz.logback.kafka.KafkaAppender">
        <topic>mytopic</topic>
        <zookeeperHost>localhost:2181</zookeeperHost>
    </appender>
    <root level="debug">
        <appender-ref ref="KAFKA" />
    </root>
</configuration>
```

我们开发的这个 Storm 应用是对时间敏感的：如果追踪事件发生的频率，我们需要知道每个事件发生的时间。一个自然的实现方法是给每个事件用 System.currentTimeMillis() 在每个事件进入 topology 时生成一个时间戳。然而，Trident 的批处理机制不能保证所有的 tuple 都按照其发射的速率来接收。

为了解决这种情况，我们需要捕获事件发生的时间并且在将数据写到 Kafka 队列的时候将时间戳写入数据中。幸运的是，ILoggingEvent 类本身已经自带了一个时间戳记录事件发生时间，精度是微秒。

为了将这个元数据包括到 ILoggingEvent，我们会建立一个自定义的 Formatter 的实现将日志事件数据编码为 JSON 格式：

```
public class JsonFormatter implements Formatter {
    private static final String QUOTE = "\"";
    private static final String COLON = ":";
    private static final String COMMA = ",";

    private boolean expectJson = false;

    public String format(ILoggingEvent event) {
        StringBuilder sb = new StringBuilder();
        sb.append("{");
        fieldName("level", sb);
        quote(event.getLevel().levelStr, sb);
        sb.append(COMMA);
        fieldName("logger", sb);
        quote(event.getLoggerName(), sb);
        sb.append(COMMA);
        fieldName("timestamp", sb);
        sb.append(event.getTimeStamp());
        sb.append(COMMA);
        fieldName("message", sb);
        if (this.expectJson) {
            sb.append(event.getFormattedMessage());
        } else {
            quote(event.getFormattedMessage(), sb);
        }

        sb.append("}");
        return sb.toString();
    }

    private static void fieldName(String name, StringBuilder sb) {
        quote(name, sb);
        sb.append(COLON);
    }
```

```
    private static void quote(String value, StringBuilder sb) {
        sb.append(QUOTE);
        sb.append(value);
        sb.append(QUOTE);
    }

    public boolean isExpectJson() {
        return expectJson;
    }

    public void setExpectJson(boolean expectJson) {
        this.expectJson = expectJson;
    }
}
```

JsonMessageFormatter 类的代码段使用了 java.lang.StringBuilder 类来从 ILoggingEvent 对象建立 JSON 数据。当然我们也可以使用 JSON 库来做这件事，因为我们生成的 JSON 数据很简单，为这点需求没有必要添加一个 JSON 类库的依赖。

JsonMessageFormatter 使用 JavaBean 中的一个属性 expectJson，这个布尔变量用来指明，传输给 Formatter 实现的数据是否应当作为 JSON 数据对待。如果设置为 False，则日志消息会作为双引号的字符串，否则消息会作为 JSON 对象（{…}）或者数组（[…]）。

下面的代码是 logback 配置实例，说明如何使用 KafkaAppender 类和 JsonFormatter 类：

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <appender name="KAFKA"
        class="com.github.ptgoetz.logback.kafka.KafkaAppender">
        <topic>foo</topic>
        <zookeeperHost>localhost:2181</zookeeperHost>
        <!-- specify a custom formatter -->
        <formatter class="com.github.ptgoetz.logback.kafka.formatter.
JsonFormatter">
            <!--
            Whether we expect the log message to be JSON encoded or
not.
            If set to "false", the log message will be treated as
a string, and wrapped in quotes. Otherwise it will be treated as a
parseable JSON object.
            -->
            <expectJson>false</expectJson>
        </formatter>
    </appender>
    <root level="debug">
        <appender-ref ref="KAFKA" />
    </root>
</configuration>
```

因为我们建立的分析 topology 比较关心事件的时间而不是消息内容，我们生成的日志消息会是字符串，因此将 expectJson 属性设置为 False。

4.5 日志分析 topology

有了将日志写到 Kafka 的途径，我们现在可以将注意力集中到实现 Trident topology 以进行分析计算了。topology 会进行下面的操作：

1. 接收并且解析 JSON 格式的原始日志数据。
2. 提取并且发射需要的字段。
3. 更新指数移动平均值 function。
4. 判断移动平均值是否超过了特定的阈值。
5. 过滤除了状态变更之外的事件（例如，在阈值以上或者以下的数据）。
6. 发送即时消息（XMPP）通知。

这个 topology 由图 4-2 说明，Trident 数据流操作在上面，数据流处理的组件在下面。

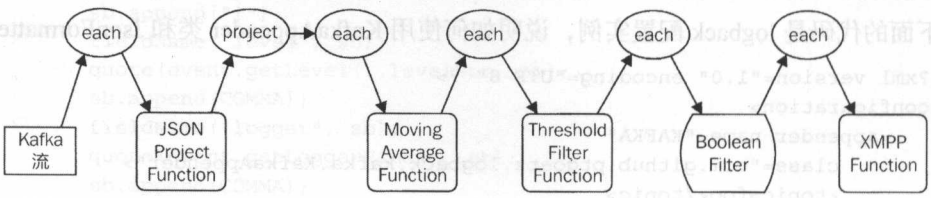


图 4-2

4.5.1 Kafka spout

建立日志分析 topology 的第一步是配置 Kafka spout 从 Kafka 里读取数据流到我们的 topology 中：

```
        TridentTopology topology = new TridentTopology();

        StaticHosts kafkaHosts = KafkaConfig.StaticHosts.
fromHostString(Arrays.asList(new String[] { "localhost" }), 1);
        TridentKafkaConfig spoutConf = new
TridentKafkaConfig(kafkaHosts, "log-analysis");
        spoutConf.scheme = new StringScheme();
        spoutConf.forceStartOffsetTime(-1);
        OpaqueTridentKafkaSpout spout = new OpaqueTridentKafkaSpout(s
```

```
poutConf);

        Stream spoutStream = topology.newStream("kafka-stream",
spout);
```

这些代码首先建立了一个 TridentTopology 实例，然后使用 Kafka Java API 建立了一个待连接的 Kafka 主机（因为我们使用了单节点，非集群的本地 Kafka 服务，我们指定了一个主机：localhost）。下一步，建立一个 TridentKafkaConfig 对象，将主机列表和唯一标识符传递给这个对象。

我们写给 Kafka 队列的数据就是简单的 Java 字符串，因此我们使用 Storm-Kafka 内置的 StringScheme 类。StringScheme 类会从 Kafaka 中读取数据，作为字符串处理并在 tuple 中输出一个叫 str 的字段。默认情况下，当 Kafka 队列部署时会先查找 ZooKeeper 的状态信息，尝试从上次读取的位置继续读取数据。这个特性可以通过调用 TridentKafkaConfig 类中的 forceOffsetTime（long time）方法来强制覆盖。时间参数可以是以下三个值中的一个：

- -2（最早的位置）：spout 会回退（rewind）并且从队列开始读取数据。
- -1（最后的位置）：spout 会快进（fast forward）并且从队列末尾开始读取数据。
- 微秒时间戳：给出一个指定的时间（比如 java.util.Date.getTime()），spout 会从这个时间点开始读取数据。

在设置好 spout 配置之后，我们会建立一个不透明事务型的 Kafka spout，并且配置一个对应的 Trident 数据流。

4.5.2　JSON project function

从 Kafka spout 发送出来的数据流只包括一个字段（str），其中包括了日志事件的 JSON 数据。我们要建立一个 Trident function 来解析读入的数据并且输出，或者作为 tuple 的值域来发射需求的字段，如下面代码所示：

```
public class JsonProjectFunction extends BaseFunction {

    private Fields fields;

    public JsonProjectFunction(Fields fields) {
        this.fields = fields;
    }

    public void execute(TridentTuple tuple, TridentCollector
collector) {
```

```
            String json = tuple.getString(0);
            Map<String, Object> map = (Map<String, Object>)
                JSONValue.parse(json);
            Values values = new Values();
            for (int i = 0; i < this.fields.size(); i++) {
                values.add(map.get(this.fields.get(i)));
            }
            collector.emit(values);
        }
    }
```

JsonProjectFunction 的构造函数使用了一个 Fields 对象作为参数，这个参数决定了从 JSON 数据中要查找哪个字段的数据向外发射。当这个 function 接收到一个 tuple，首先解析 JSON 数据数据中“str”字段的值，遍历 Fields 对象中的值，并且从 JSON 中提取并发射对应字段的值。

下列代码生成一个 Fields 对象，其中包括了需要从 JSON 中提取的字段名列表。然后从 spout 的数据流中建立一个新的 Stream 对象，选取 tuple 中的 str 字段作为 JsonProjectFunction 构造函数的输入，运行构造函数，指定从 JSON 中提取的字段，也是 funciton 输出的字段：

```
        Fields jsonFields = new Fields("level", "timestamp",
"message", "logger");
        Stream parsedStream = spoutStream.each(new Fields("str"), new
JsonProjectFunction(jsonFields), jsonFields);
```

考虑下面这条 JSON 数据被 Kafka spout 读取后：

```
{
  "message" : "foo",
  "timestamp" : 1370918376296,
  "level" : "INFO",
  "logger" : "test"
}
```

这个 function 会输出下列 tuple：

```
[INFO, 1370918376296, test, foo]
```

4.5.3 计算移动平均值

为了计算事件发生的频率，不需要存储过多的状态数据，我们实现一个 function 用来执行指数移动平均值的统计计算。

一个移动平均值计算常常用来在事件序列数据中消除短期波动，展示长期的趋势。移动平均值最常见的一个例子是在图示股票市场中股票波动，如图 4-3 所示。

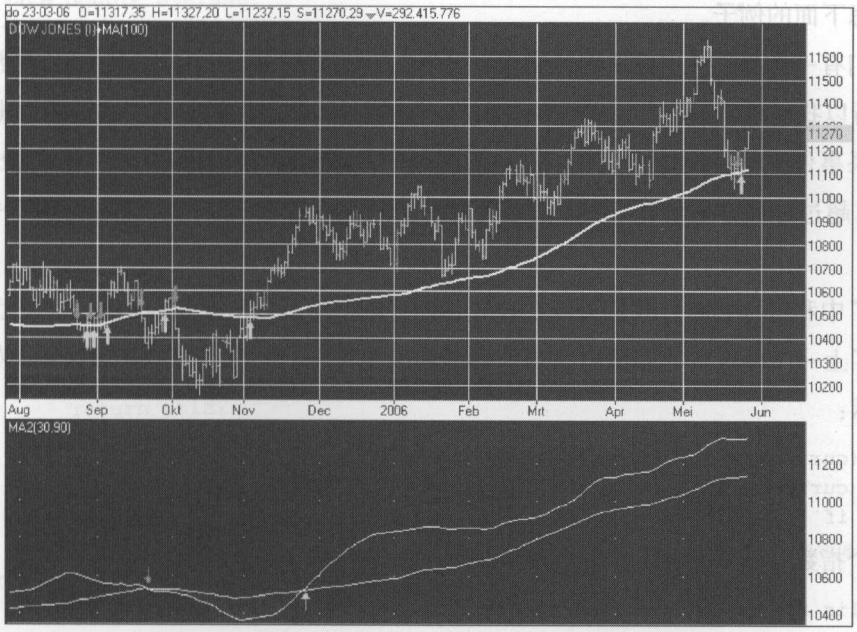


图　4-3

移动平均值的平滑效果通过在计算中考虑到历史值来实现。计算一个移动平均值可以通过少量的状态来进行。对于一个事件序列，我们只需要记录上次发生的时间和上次计算出来的平均值即可。

在伪代码如下所示：

```
diff = currentTime - lastEventTime
currentAverage = (1.0 - alpha) * diff + alpha * lastAverage
```

上述计算中的 alpha 值是一个 0 到 1 之间的常量。alpha 值决定了一段时间内的平滑水平。alpha 值越趋于 1，历史值对当前的平均值的影响越大。换句话说，alpha 值越接近 0，结果平滑度越低，移动平均数越趋近于当前的值。趋近于 1 的 alpha 值会有相反的作用。当前的平均值会具有更平滑的波动，在决定当前平均值时，历史值会具有更多的权重。

4.5.4　添加一个滑动窗口

某些情况下，我们需要降低历史值对当前移动平均值的影响，例如，当两次事件之间

间隔时间较长时，需要重置平滑作用。如果有一个较小的 alpha 值，可能不需要这么做，因为平滑效果已经很小。但是，当如果 alpha 值很大时，需要适当地降低平滑效果的影响。

考虑下面的例子。

我们有一个事件（比如说网络错误）很少发生。偶尔出现小的峰值，通常是没什么问题的。所以我们想平滑这些小的峰值。只有当连续的峰值出现时，我们才需要发出通知。

如果事件平均一周才发生一次（达不到通知的阈值），但是某一天一小时内出现了多个峰值（超过了通知阈值），alpha 值较大的平滑效果可能抵消了峰值，导致事件一直无法触发。

为了中和这种影响，我们可以在计算移动平均值时引入滑动窗口的概念。因为我们已经保留了上一个事件的时间戳以及当前的平均值，实现一个滑动窗口非常简单，如下面伪代码所示：

```
if (currentTime - lastEventTime) > slidingWindowInterval
    currentAverage = 0
end if
```

一个指数移动平均值计算的代码实现如下所示：

```
public class EWMA implements Serializable {

    public static enum Time {
        MILLISECONDS(1), SECONDS(1000), MINUTES(SECONDS.getTime() *
60), HOURS(MINUTES.getTime() * 60), DAYS(HOURS
                .getTime() * 24), WEEKS(DAYS.getTime() * 7);

        private long millis;

        private Time(long millis) {
            this.millis = millis;
        }

        public long getTime() {
            return this.millis;
        }
    }

    // Unix load average-style alpha constants
    public static final double ONE_MINUTE_ALPHA = 1 - Math.exp(-5d /
60d / 1d);
    public static final double FIVE_MINUTE_ALPHA = 1 - Math.exp(-5d /
60d / 5d);
    public static final double FIFTEEN_MINUTE_ALPHA = 1 - Math.exp(-5d
/ 60d / 15d);
```

```
private long window;
private long alphaWindow;
private long last;
private double average;
private double alpha = -1D;
private boolean sliding = false;

public EWMA() {
}

public EWMA sliding(double count, Time time) {
    return this.sliding((long) (time.getTime() * count));
}

    public EWMA sliding(long window) {
        this.sliding = true;
        this.window = window;
        return this;
    }

    public EWMA withAlpha(double alpha) {
        if (!(alpha > 0.0D && alpha <= 1.0D)) {
            throw new IllegalArgumentException("Alpha must be between
0.0 and 1.0");
        }
        this.alpha = alpha;
        return this;
    }

    public EWMA withAlphaWindow(long alphaWindow) {
        this.alpha = -1;
        this.alphaWindow = alphaWindow;
        return this;
    }

    public EWMA withAlphaWindow(double count, Time time) {
        return this.withAlphaWindow((long) (time.getTime() * count));
    }

    public void mark() {
        mark(System.currentTimeMillis());
    }

    public synchronized void mark(long time) {
        if (this.sliding) {
            if (time - this.last > this.window) {
                // reset the sliding window
                this.last = 0;
```

```
            }
        }
        if (this.last == 0) {
            this.average = 0;
            this.last = time;
        }
        long diff = time - this.last;
        double alpha = this.alpha != -1.0 ? this.alpha : Math.exp(-1.0
* ((double) diff / this.alphaWindow));
        this.average = (1.0 - alpha) * diff + alpha * this.average;
        this.last = time;
    }

    public double getAverage() {
        return this.average;
    }

    public double getAverageIn(Time time) {
        return this.average == 0.0 ? this.average : this.average /
time.getTime();
    }

    public double getAverageRatePer(Time time) {
        return this.average == 0.0 ? this.average : time.getTime() /
this.average;
    }

}
```

EWMA 的实现定义了三个 alpha 的常量：ONE_MINUTE_ALPHA、FIVE_MINUTE_ALPHA、FIFTEEN_MINUTE_ALPHA。这些值和 UNIX 系统计算负载时使用的标准 alpha 值相同。Alpha 值也可以手工指定，或者作为一个 alpha 窗口的函数。

实现中使用了流式风格的 bulder API。例如，你可以建立一个 EWMA 实例使用一个 1 分钟的滑动窗口，alpha 值和 UNIX 中一分钟的值相同，代码如下：

```
EWMA ewma = new EWMA().sliding(1.0, Time.MINUTES).withAlpha(EWMA.ONE_
MINUTE_ALPHA);
```

mark() 方法用来更新移动平均值。没有参数的话，mark() 方法会使用当前时间来计算平均值。因为我们需要使用日志事件发生时的时间戳来进行计算，所以我们重写了 mark() 方法使它可以接受指定的事件。

getAverage() 方法返回 mark() 方法多次调用的平均间隔时间，单位是微秒。我们还增加了一个 getAverageIn() 方法，会按照特定的时间单位来返回平均值（秒、分钟、小时，

等等）。GetAverageRatePer() 方法返回特定时间度量内调用 mark() 的频率。

你可能已经注意到了，使用指数滑动平均可能会比较棘手。寻找合适的 alpha 值和对应的滑动窗口的值会依赖于不同的使用场景，找到适当的值是一个试错的过程。

4.5.5　实现滑动平均 function

为了实现 Trident topology 中使用 EWMA 类，我们建立一个 Trident 的 BaseFunction 抽象类的子类，叫做 MovingAverageFunction，会包装一个 EWMA 实例，如下面代码所示：

```
public class MovingAverageFunction extends BaseFunction {
    private static final Logger LOG = LoggerFactory.
getLogger(BaseFunction.class);

    private EWMA ewma;
    private Time emitRatePer;

    public MovingAverageFunction(EWMA ewma, Time emitRatePer){
        this.ewma = ewma;
        this.emitRatePer = emitRatePer;
    }

    public void execute(TridentTuple tuple, TridentCollector
collector) {
        this.ewma.mark(tuple.getLong(0));
        LOG.debug("Rate: {}", this.ewma.getAverageRatePer(this.
emitRatePer));
        collector.emit(new Values(this.ewma.getAverageRatePer(this.
emitRatePer)));
    }
}
```

MovingAverage.execute() 方法从读入的 tuple 的第一个字段中获取 Long 的值，使用这个值调用 mark() 方法来更新当前的平均速率，并且发送当前的平均值。Trident 的 Function 只能添加数据，以为这需要在数据流里的 tuple 中添加字段。例如，考虑下面这个 tuple 进入了这个 function：

```
[INFO, 1370918376296, test, foo]
```

处理完成后，这个 tuple 可能会像下面这样：

```
[INFO, 1370918376296, test, foo, 3.72234]
```

这里，新的字段值表示了平均速率。

为了使用这个 funciton，我们建立一个 EWMA 类，并且将其发送给 MovingAverage-

Function 构造函数。将这个 function 和 each() 方法一起使用，选择 timestamp 字段作为输入。代码如下所示：

```
        EWMA ewma = new EWMA().sliding(1.0, Time.MINUTES).
withAlpha(EWMA.ONE_MINUTE_ALPHA);
        Stream averageStream = parsedStream.each(new
Fields("timestamp"),
                new MovingAverageFunction(ewma, Time.MINUTES), new
Fields("average"));
```

4.5.6 按照阈值进行过滤

我们的使用场景中，会定义一个速率的阈值，当超过阈值时触发事件通知。当平局速率跌至阈值以下时（意味着恢复正常）也会发送一个事件通知。我们通过另外一个 function 和一个简单 Trident 过滤器来实现这个功能。

这个 function 的工作是当前 tuple 的平均速率字段的值是否超过了阈值，以及这个值和前一个值相比是否发生了状态变化（意味着，速率值是否从阈值一下变成了阈值以上，或者相反）。如果一个新的平均值表示了一个状态的变化，这个 function 会发送一个布尔值 True，否则，会发送一个 False。我们会利用这个值来过滤掉那些不引起状态变化的事件。我们将在 ThresholdFilterFunction 类中实现阈值追踪 function。代码片段如下：

```
public class ThresholdFilterFunction extends BaseFunction {
    private static final Logger LOG = LoggerFactory.getLogger(Threshol
dFilterFunction.class);

    private static enum State {
        BELOW, ABOVE;
    }

    private State last = State.BELOW;
    private double threshold;

    public ThresholdFilterFunction(double threshold){
        this.threshold = threshold;
    }

    public void execute(TridentTuple tuple, TridentCollector
collector) {
        double val = tuple.getDouble(0);
        State newState = val < this.threshold ? State.BELOW : State.
ABOVE;
        boolean stateChange = this.last != newState;
```

```
            collector.emit(new Values(stateChange, threshold));
            this.last = newState;
            LOG.debug("State change? --> {}", stateChange);
        }
    }
```

ThresholdFilterFunction 类定义了一个内部枚举用来表达状态（在阈值之上或者之下）。构造函数读入一个 double 参数用来确定对比的阈值。在 execute() 方法中，我们获取当前的速率值，判断它是在阈值之下还是智商。然后将它和上一个状态进行对比，如果有状态的变化则发射一个布尔值。最后用新计算出来的值更 function 内部的状态是在阈值之上 / 或之下。

在经过了 ThresholdFilterFunction 类的处理后，输入数据流的 tuple 会增加一个新的布尔值字段，我们可以用来过滤那些没有触发状态变化的事件。为了找出非状态变化的事件，我们将使用一个 BoolenFilter 类，如下所示：

```
public class BooleanFilter extends BaseFilter {

    public boolean isKeep(TridentTuple tuple) {
        return tuple.getBoolean(0);
    }
}
```

BooleanFilter.isKeep() 方法简单的读入 tuple 中特定字段的布尔值，并且返回这个字段。任何包括 False 值的 tuple 会在结果数据流中过滤掉。

下面代码片段示例如何使用 ThresholdFilterFuncation 类和 BooleanFilter 类：

```
        ThresholdFilterFunction tff = new
ThresholdFilterFunction(50D);
        Stream thresholdStream = averageStream.each(new
Fields("average"), tff, new Fields("change", "threshold"));
        Stream filteredStream = thresholdStream.each(new
Fields("change"), new BooleanFilter());
```

第一行建立了一个 ThresholdFilterFunction 实例，设置阈值为 50.0。我们然后建立一个数据流使用 averageStream 作为输入调用阈值 function，然后选择 tuple 中的 average 字段作为输入。我们还指定了 function 添加的字段的名称（change 和 threshold）。最后，我们应用 BooleanFilter 类来建立一个新的数据流，值包含了有状态变更的 tuples。

这时候，实现通知告警的所有需求已经满足了。建立的 filteredStream 数据流值包括出现状态变化的 tuples。

4.5.7 通过 XMPP 发送通知

XMPP 协议提供了即时通信标准中期望的特性：

- 通信录（联系人列表）
- 在线状态（可以知道别人上线或者其他状态）
- 用户之间及时消息
- 分组聊天

XMPP 协议使用 XML 格式作为通信协议，但是有很多中高级客户端类库提供了简单的 API，屏蔽了底层实现细节。我们将使用 Smack API（http://www.igniterealtime.org/projects/smack/），因为它是最简单的 XMPP 客户端实现之一。

下面代码片段示范了使用 Smack API 发送一个简单的即时消息到其他用户：

```
// connect to XMPP server and login
ConnectionConfiguration config = new
    ConnectionConfiguration("jabber.org");
XMPPConnection client = new XMPPConnection(config);
client.connect();
client.login("username", "password");

// send a message to another user
Message message =
    new Message("myfriend@jabber.org", Type.normal);
message.setBody("How are you today?");
client.sendPacket(message);
```

这段代码连接到 jabber.org 的 XMPP 服务器，并且使用用户名和密码登录。这个场景之下，Smack 库处理了和服务器的底层通信。当客户端连接并且登录认证之后，它会发送一个在场的消息到 server。这允许用户的联系人（XMPP 通信录中列出的用户）收到一个通知这个用户上线了。最后，会建立和发送一个简单的消息到“myfriend@jabber.org”。

基于这个简单的例子，我们建立一个叫做 XMPPFunction 的类，它会在接收到 Trident tuple 时发送 XMPP 通知。这个类会在 prepare() 方法中确立一个到 XMPP 服务器的长连接。在 execute() 方法中会基于收到的 tuple 建立一个 XMPP 消息。

为了让 XMPPFunction 类复用性更强，我们介绍 MessageMapper 接口来定一个方法从 Trident tuple 中格式化数据到适用于消息通知的字符串。代码示例如下：

```
public interface MessageMapper extends Serializable {
    public String toMessageBody(TridentTuple tuple);
}
```

XMPPFunction 类中，我们会将消息格式化操作委托给一个 MessageMapper 实例。代码如下：

```
public class XMPPFunction extends BaseFunction {
    private static final Logger LOG = LoggerFactory.
getLogger(XMPPFunction.class);

    public static final String XMPP_TO = "storm.xmpp.to";
    public static final String XMPP_USER = "storm.xmpp.user";
    public static final String XMPP_PASSWORD = "storm.xmpp.password";
    public static final String XMPP_SERVER = "storm.xmpp.server";

    private XMPPConnection xmppConnection;
    private String to;
    private MessageMapper mapper;

    public XMPPFunction(MessageMapper mapper) {
        this.mapper = mapper;
    }

    @Override
    public void prepare(Map conf, TridentOperationContext context) {
        LOG.debug("Prepare: {}", conf);
        super.prepare(conf, context);
        this.to = (String) conf.get(XMPP_TO);
        ConnectionConfiguration config = new ConnectionConfiguration((
String) conf.get(XMPP_SERVER));
        this.xmppConnection = new XMPPConnection(config);
        try {
            this.xmppConnection.connect();
            this.xmppConnection.login((String) conf.get(XMPP_USER),
(String) conf.get(XMPP_PASSWORD));
        } catch (XMPPException e) {
            LOG.warn("Error initializing XMPP Channel", e);
        }
    }

    public void execute(TridentTuple tuple, TridentCollector
collector) {
        Message msg = new Message(this.to, Type.normal);
        msg.setBody(this.mapper.toMessageBody(tuple));
        this.xmppConnection.sendPacket(msg);

    }
}
```

XMPPFunction 类首先定义了几个字符串常量，用来从 Storm 配置中查询配置选项的

值传递给 prepare() 方法。后面就是一些实例变量的声明，会在 function 激活时进行赋值。类的构造函数使用了一个 MessageMapper 实例作为参数，这个参数会在 execute() 方法中用来格式化通知消息的包体。

在 prepare() 方法中，我们为 XMPPConnection 类查找配置参数（server、username、to address 等），并且建立连接。当使用了这个 function 的 topology 部署时，XMPP client 会发送在线的数据包，在通信录里添加了这个用户的其他用户会接受到这个用户的上线通知。

我们的通知机制里最后需要的一部分是实现一个 MessageMapper 实例，将 tuple 的内容格式化为可供人阅读的消息体，代码如下：

```
public class NotifyMessageMapper implements MessageMapper {

    public String toMessageBody(TridentTuple tuple) {
        StringBuilder sb = new StringBuilder();
        sb.append("On " + new Date(tuple.getLongByField("timestamp"))
+ " ");
        sb.append("the application \"" + tuple.
getStringByField("logger") + "\" ");
        sb.append("changed alert state based on a threshold of " +
tuple.getDoubleByField("threshold") + ".\n");
        sb.append("The last value was " + tuple.
getDoubleByField("average") + "\n");
        sb.append("The last message was \"" + tuple.
getStringByField("message") + "\"");
        return sb.toString();
    }
}
```

4.6 最终的 topology

我们已经实现了建立 log 分析 topology 所需的所有组件：

```
public class LogAnalysisTopology {

    public static StormTopology buildTopology() {
        TridentTopology topology = new TridentTopology();

        StaticHosts kafkaHosts = KafkaConfig.StaticHosts.
fromHostString(Arrays.asList(new String[] { "localhost" }), 1);
        TridentKafkaConfig spoutConf = new
TridentKafkaConfig(kafkaHosts, "log-analysis");
        spoutConf.scheme = new StringScheme();
        spoutConf.forceStartOffsetTime(-1);
```

```
        OpaqueTridentKafkaSpout spout = new OpaqueTridentKafkaSpout(s
poutConf);

        Stream spoutStream = topology.newStream("kafka-stream",
spout);
        Fields jsonFields = new Fields("level", "timestamp",
"message", "logger");
        Stream parsedStream = spoutStream.each(new Fields("str"), new
JsonProjectFunction(jsonFields), jsonFields);

        // drop the unparsed JSON to reduce tuple size
        parsedStream = parsedStream.project(jsonFields);

        EWMA ewma = new EWMA().sliding(1.0, Time.MINUTES).
withAlpha(EWMA.ONE_MINUTE_ALPHA);
        Stream averageStream = parsedStream.each(new
Fields("timestamp"),
                new MovingAverageFunction(ewma, Time.MINUTES), new
Fields("average"));

        ThresholdFilterFunction tff = new
ThresholdFilterFunction(50D);
        Stream thresholdStream = averageStream.each(new
Fields("average"), tff, new Fields("change", "threshold"));

        Stream filteredStream = thresholdStream.each(new
Fields("change"), new BooleanFilter());

        filteredStream.each(filteredStream.getOutputFields(), new
XMPPFunction(new NotifyMessageMapper()), new Fields());

        return topology.build();
    }

    public static void main(String[] args) throws Exception {
        Config conf = new Config();
        conf.put(XMPPFunction.XMPP_USER, "storm@budreau.local");
        conf.put(XMPPFunction.XMPP_PASSWORD, "storm");
        conf.put(XMPPFunction.XMPP_SERVER, "budreau.local");
        conf.put(XMPPFunction.XMPP_TO, "tgoetz@budreau.local");

        conf.setMaxSpoutPending(5);
        if (args.length == 0) {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("log-analysis", conf,
buildTopology());
        } else {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopology(args[0], conf,
```

```
        buildTopology());
            }
        }
    }
```

然后，buildTopology() 方法建立了所有 Kafka spout 到 Trident function 和 filter 之间的数据流连接。通过 main() 方法提交 topology 到一个集群：如果 topology 在本地模式下运行就是一个本地集群，或者在远程集群模式下分布式运行。

我们首先配置 Kafka spout 从 topic 读取数据，topic 和我们应用写入 log 的 topic 保持一致。因为 Kafka 队列持久话存储所有收到的消息，还因为我们的程序会运行一段时间（可能记录很多事件），我们通过执行 forceStartOffsetTime() 方法，将参数设置为 -1，让 spout 使用 fast-forward 模式直接去读取当前 Kafka 队列中的最后一个记录。这样会避免将所有消息重放，我们并不关心过期消息。使用 -2 作为参数会强制 spout 从队列的开头重新读取数据，使用特定的微秒值会指定重放特定时间后的数据。如果没有调用 forceFromStartTime() 方法，spout 会尝试从 ZooKeeper 中读取上次读取中断的位置，并从该点恢复读取数据。

下一步，建立 JsonProjectFunction 类来解析从 Kafka 队列中读取的原始 JSON 数据，并且发射我们感兴趣的字段。记住，Trident 的 function 只能添加数据。意味着我们的所有 tuple 数据流，都是在 JSON 提取出的字段值的基础上添加的，都会包含着原始的未解析的 JSON 字符串。因为我们不在使用这个数据，我们提供我们想保留的字段列表，调用 Stream.project() 方法。Project() 方法可以削减 tuple 数据流，只保留必需的字段，当重新分片的数据量很大时这个功能非常重要。

返回的数据流现在仅包含了我们需要的数据。建立 1 分钟滑动窗口 EWMA 实例，并且配置 MovingAverageFunction 类发送当前一分钟的速率。再建立阈值为 50.0 的 ThresholdFunction 类，在任何时间出现平均速率超过 50 或跌至每分钟 50 以下的事件，我们都会收到通知。

最后，我们应用 BooleanFilter 类并将结果数据流连接到 XMPPFunction 类。

Topology 的 main() 方法在 Config 对象中填入 XMPPFuncton 类所需要的属性，然后提交 topology。

4.7 运行日志分析 topology

为了运行分析 topology，首先按照本章先前列出的概述，保证 ZooKeeper，Kafka 和

OpenFire 服务都在正常运行。然后运行 topology 的 main() 方法。

当 topology 激活时，storm XMPP 用户会连接到 XMPP 服务器，并且触发一个在线事件。如果你也使用客户端登录了同一个服务器，并且将 storm 用户加入了通信录，你会看到 storm 用户在线状态。如下面图 4-4 所示。

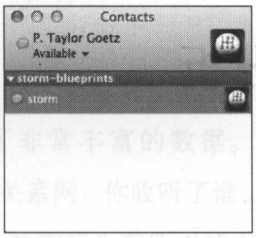


图　4-4

下一步，运行 RogueApplication 类并且等待一分钟。你会收到一个及时消息通知表明阈值被超过了，跟随其后的一条消息表明已经恢复正常（低于阈值），如图 4-5 所示。

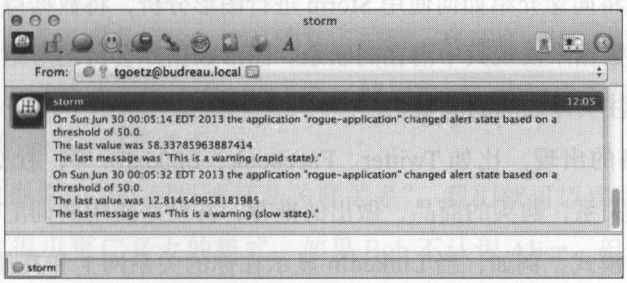


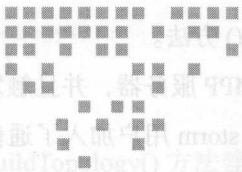
图　4-5

总结

本章中，我们通过建立一个简单但强大的 topology 向您介绍了实时分析，这个 topology 稍加改动就可以应用到多种场景下。我们建立的组件都是通用的，在其他项目里很容易就可以复用和扩展。最后，我们介绍了一种实际场景下的 spout 实现，适用于多种目的。

实时分析的话题很宽泛，本章显然只对其做了简单的介绍，我们鼓励读者进一步研究上述及其他章节的技术，考虑如何让这些技术整合到你的分析工具箱中。

下一章，通过建立一个向图形数据库持续写入数据的应用，我们将介绍 Trident 的分布式状态机制。



Chapter 3 第 5 章

实时图形分析

本章中，我们将向您介绍如何使用 Storm 进行图形分析，将数据持久化存储在图形数据库中，并且查询数据来发现其中潜在的联系。图形数据库会将图形结构用顶点、边属性进行存储，主要关注实体之间的关系。

随着社交网络的出现，比如 Twitter、Facebook、LinkedIn 等，社交图形变得无所不在。分析人之间的关系，购买的商品，做出的推荐，甚至是使用的词汇都可以提取出不同于传统数据模型的模式。例如，当 LinkedIn 显示在你的关系网上和其他人的距离是四格，Twitter 向你推荐收听列表，或者 Amazon 向你推荐可能感兴趣的商品，这都是利用了他们获取的你的关系图。图形数据库就是用于对这些关系进行分析。

在本章中，我们构建一个应用，使用 Twitter 公共数据的一个子集（Twitter 用户消息的实时推荐 feed），在图形数据库中基于每条数据的消息内容，建立节点（顶点）和关系（边），供后续分析使用。Twitter 中最明显的关系就是用户之间的收听 / 被收听关系，我们可以在这些明显的关系之上，推测出更深层的关系。通过查看消息内容，我们可以使用消息元数据（# 标签、用户提及等）来识别，比如说哪些用户提及了相同的话题或者发表了相同标签的消息。在本章中，我们将讲到下列话题：

- 基础的图形数据库概念。
- TinkerPop 图形 API。

- 图形数据建模。
- 和 Titan 分布式图形数据库交互。
- 后端基于图形数据开发一个 Trident 状态的实现。

5.1　使用场景

当前的社交网站已经获取了非常丰富的数据。很多社交媒体服务比如 Twitter、Facebook、LinkedIn 都是主要基于关系网：你收听了谁，朋友是哪些人，和谁有工作关系。在这些表层显而易见的关系之下，社交行为理所当然也会具有一些持久和隐藏的关系链。以 Twitter 为例，明显的关系包括每个人收听和被收听的人。不太明显的关系是在使用服务中可能是无意间建立的。你是否在 Twitter 上向别人转发过消息？如果是，你就建立了一个关系。发表消息中包括一个 URL 链接？如果是，又建立了一个关系。在 Facebook 中连接了一个产品，服务，或者评论？这又建立了关系。甚至在 tweet 消息使用了某些特定单词或者成语，都可以被认为建立了一种关系。使用单词，你形成了和单词之间的关系，重复使用该单词，就强化了这种关系。

如果我们将数据看作“任何事情都是关联关系”，我们就可以建立一个格式化的数据集，并且分析它得出更广义上的模式。如果 Bob 不认识 Alice，但是 Bob 和 Alice 都发表过某个 URL 的 tweet 消息，我们从这个事实中推出一种关系。当数据集规模增长时，关系网络中的关系数量也会增长（类似 Metcalfe 定律：http://en.wikipedia.org/wiki/Metcalfe%27s_law，网络的价值等于网络节点数的平方，网络的价值与联网用户数的平方成正比）。

当我们开始查询数据集时，会从不断增加的关系网中收集模式，图形数据库中存储数据会随之显著增长。我们执行的图形分析适用于一系列现实生活中的场景，包括下面这些列子：

- 定向广告
- 推荐引擎
- 情感分析

5.2 体系结构

我们应用的结构相对比较简单。建立一个 Twitter 客户端程序来读取 Twitter 公共信息的子集。将每个队列以 json 格式写到 kafka 队列中。然后用 kafka spout 将数据输入 storm topology。最后，storm topology 会分析读入的消息，并且将结果存在图形数据库中，见图 5-1。

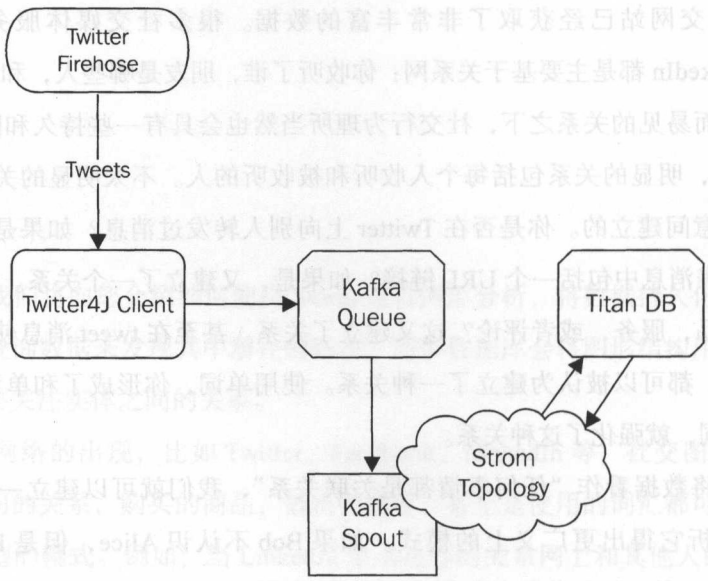


图 5-1

5.2.1 Twitter 客户端

Twitter 提供了一个完整的静态 API，除了可以提供传统的请求应答接口，还提供了支持长连接的数据流 API。Twitter4J Java 库（http://twitter4j.org/）对最新版本的 TwitterAPI 全面兼容，并且为底层细节操作（连接管理，认证，JSON 解析等）提供了纯 Java 的 API。我们将使用 Twitter4J 来连接到 Twitter 数据流 API。

5.2.2 Kafka spout

在前面的章节，我们开发了一个 Logback Appender 的插件用来方便地将数据发布到

kafka 队列中，并且在 topology 中使用了 Nathan Marz 的 Kafka spout（https://github.com/nathanmarz/storm-contrib）来消费数据。虽然使用 Twitter4J 和 Twittet 数据流 API 直接开发 spout 也很简单，但使用 Kafka 和 Kafka Spout 可以提供事务性，唯一性的语义，它内置的容错机制也省去了我们自己开发的成本。Kafka 队列的详细信息和安装使用方法见第 4 章，实时趋势分析。

5.2.3　Titan 分布式图形数据库

Titan 图形数据库会优化图形结构的存储和查询。和 Storm 和 Kafka 一样，Titan 数据库可以运行在集群模式下，并且可以水平扩展来适应数据量和用户负载的增长。Titan 可以将数据存储在三种后端：Apache　Cassandra、Apache HBase 和 Oracle Berkely Database。选择那种后端基于用户更期望 CAP 原理中的那两种属性。对于一个 NoSQL 数据库，CAP 原理规定了分布式系统不能同时保证下列三种属性：

- 一致性（Consistency）：所有客户端当前看到的都是同样的数据。
- 可用性（Availability）：部分节点故障后系统是否还能继续运行。
- 分区容错性（Partition Tolerance）：网络故障或者消息丢失系统是否能够继续运行。

在我们的例子中，一致性不是应用的关键因素。我们更关心扩展性和容错性。参考 CAP 原理三角形的图（见图 5-2），Cassandra 是适合我们的后端存储选择。

图　5-2

5.3　图形数据库简介

一个图形是对象（定点）和有向连接（边）组成的网络。图 5-3 展示了一个简单的社交图，类似于我们从 Twitter 中发现的关系。

这个例子中，用户用顶点表示（节点），关系用边表示（连接）。注意到图中的边

是有向的，这样能增加更深一层的表现力。例如，可以表达 Bob 和 Alice 相互收听的事实，Alice 收听了 Ted 但是 Ted 并没有收听 Alice。这种关系使用无向边就很难表达。

很多图形数据库遵循属性图模型。一个属性图通过增加一系列属性延伸了基本图形模型，赋值了属性的顶点和边如图 5-4 所示。

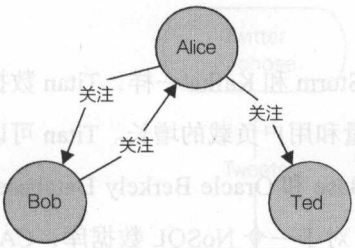


图 5-3

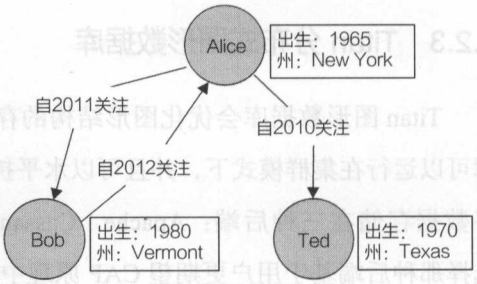


图 5-4

图模型中将属性元数据关联到对象和关系上的能力，为图形算法和查询提供了强大的支持。例如，在 Follows 的边上添加 since 属性可以使我们快速查找最近一年哪些用户收听了特定用户。

和关系型数据库对比，图形数据库中的关系表示是显示而不是隐式的。图形数据库中的关系是完备的数据结构而不仅仅是隐含的连接（如外键）。在底层，图形数据库依赖的数据结构为图形遍历操作高度优化，对非图形为主的数据模型效率通常要低一些。在关系数据模型中，遍历图形结构因为需要连接多个表，会有很高的计算成本。在图形数据库中，遍历节点是非常自然的操作。

5.3.1 访问图——TinkerPop 栈

TinkerPop 是一组集中在图形技术的开源项目，比如数据库访问、数据流、图遍历。Blueprints 是提供和属性图进行交互的通用 Java API，与 JDBC 提供给关系数据库的接口类似。它是 TinkerPop 栈的基础。栈中的其他项目都是在此基础上添加的额外功能，因此任何实现了 Blueprints API 的图形数据库都可以使用这些功能。

TinkerPop 栈包括下述几个组件（见图 5-5）：

- Blueprints：图形 API Blueprints 是一个提供了访问图数据模型的接口集合。实现了这个接口的图数据库包括 Titan、Neo4J、MongoDB 等。
- Pipes：Dataflow Processing Pipes 是一个数据流处理框架，定义和连接不同类型的数据操作，作为一个处理流图。用 Pipes 操作数据基本上和 Storm 处理数据类似。Pipes 数据流是有向无环图（Directed Acyclic Graphs，DAG），和 Storm 的 topology 很类似。

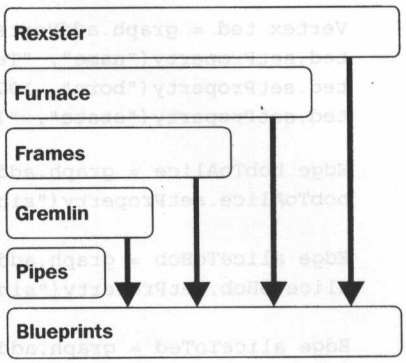


图　5-5

- Gremlin：Gremlin 是一个图遍历语言。它是一个基于 Java 的特定领域语言（Domain Specific Language，DSL），用于图形遍历、查询、分析和操作。Gremlin 发行版自带一个基于 Groovy 的 shell，允许用户分析和操作 Blueprints 图形。
- Frames：Frames 是一个对象到图形的映射框架功能类似 ORM（Object/Relation Mapping）但是为图形专门定制的。
- Furnace：Furnace 项目目的是为 Blueprints 属性图提供多种常见图形算法的实现。
- Rexster：Rexster 是图显示服务，可以将 Blueprints 图通过二进制协议 REST API 显示。

我们的目的集中在使用 Blueprints API 从 Storm topology 中生成图形数据，使用 Gremlin 进行查询和分析。

5.3.2　使用 Blueprints API 操作图形

Blueprints API 非常简单，下面的代码列出了如何通过 Blueprints API 建立前面的图形：

```
Graph graph = new TinkerGraph();

Vertex bob = graph.addVertex(null);
bob.setProperty("name", "Bob");
bob.setProperty("born", 1980);
bob.setProperty("state", "Vermont");

Vertex alice = graph.addVertex(null);
```

```
alice.setProperty("name", "Alice");
alice.setProperty("born", 1965);
alice.setProperty("state", "New York");

Vertex ted = graph.addVertex(null);
ted.setProperty("name", "Ted");
ted.setProperty("born", 1970);
ted.setProperty("state", "Texas");

Edge bobToAlice = graph.addEdge(null, bob, alice, "Follows");
bobToAlice.setProperty("since", 2012);

Edge aliceToBob = graph.addEdge(null, alice, bob, "Follows");
aliceToBob.setProperty("since", 2011);

Edge aliceToTed = graph.addEdge(null, alice, ted, "Follows");
aliceToTed.setProperty("since", 2010);

graph.shutdown();
```

第一行代码给出 com.tinkerpop.blueprints.Graph 接口的使用。这个例子中，我们建立了一个内存中的示例图（com.tinkerpop.blueprints.impls.tg.TinkerGraph）来研究。后面，我们会演示如何连接到分布式图形数据库上。

读者可能好奇，为什么把 null 作为第一个参数传递给 addVertex() 和 addEdge() 方法。这个参数是让底层的 Blueprints 实现将唯一的 ID 赋给对象的。传递 null 的意思是，让底层实现自己分配 ID 给新对象。

5.3.3 通过 Gremlin shell 操作图形

Gremlin 是一个高层的 Java API，构建在 Pipes 和 Blueprints API 基础上的。在 Java API 之外，Gremlin 还提供了基于 Groovy 的 API 和交互式的 shell（或 REPL）允许用户直接和 Blueprints 图形交互。Gremelin shell 允许建立和（或）连接并查询任何 Blueprints 图。下列代码示例了操作 Gremlin shell 的过程：

```
./bin/gremlin.sh

         \,,,/
         (o o)
-----oOOo-(_)-oOOo-----
gremlin>
gremlin> g.V('name', 'Alice').outE('Follows').count()
==>2
```

在查询图形之外，很容易使用 Gremlin 来建立和操作图形。下列代码首先会建立一个和前面例子一样的图形，这些 Groovy 代码和前面的 Java 代码作用相同：

```
g = new TinkerGraph()
bob = g.addVertex()
bob.name = "Bob"
bob.born = 1980
bob.state = "Vermont"
alice = g.addVertex()
alice.name = "Alice"
alice.born=1965
alice.state = "New York"
ted = g.addVertex()
ted.name = "Ted"
ted.born = 1970
ted.state = "Texas"
bobToAlice = g.addEdge(bob, alice, "Follows")
bobToAlice.since = 2012
aliceToBob = g.addEdge(alice, bob, "Follows")
aliceToBob.since = 2011
aliceToTed = g.addEdge(alice, ted, "Follows")
aliceToTed.since = 2010
```

在我们建立 topology 生成图形和分析图形数据的时候，你会更深入地了解如何使用 Gremlin API 和 DSL。

5.4　软件安装

我们建立的这个应用会使用到 Apache Kafka 以及它的依赖（Apache ZooKeeper）。如果还没有安装这些，按照第 2 章中的指引安装 ZooKeeper，按照第 4 章中的指引安装 Kafka。

安装 Titan

安装 Titan，首先在 https://github.com/thinkaurelius/titan/wiki/Downloads 下载 Titan 0.3.x 的完整安装包，用下面命令解压缩到适当目录：

```
wget http://s3.thinkaurelius.com/downloads/titan/titan-all-0.3.2.zip
unzip titan-all-0.3.2.zip
```

Titan 的完整发行包包括了它运行所需要的全部支持的存储后端：Cassandra、HBase 和 BerkelyDB。如果你只想使用某个特定的后端存储方式，也可以下载对应后端的发行版本。

注意 Storm 和 Titan 都使用了 Kryo 库 https://code.google.com/p/kryo/ 来进行 Java 对象的序列化。在本书写作时，Storm 和 Titan 使用了不同版本的 Kryo 库，它们联合使用时可能引发问题。

为了让 Storm 和 Titan 之间的序列化能够正常执行，需要用 Storm 发行版本中的 kryo.jar 包覆盖了 Titan 中的版本：

```
cd titan-all-0.3.2/lib
rm kryo*.jar
cp $STORM_HOME/lib/kryo*.jar ./
```

这时候，可以使用 Gremlin shell 来测试 Titan 是否安装完成：

```
$ cd titan
$ ./bin/gremlin.sh
          \,,,/
         (o o)
-----oOOo-(_)-oOOo-----
gremlin> g = GraphOfTheGodsFactory.create('/tmp/storm-blueprints')
==>titangraph[local:/tmp/storm-blueprints]
gremlin> g.V.map
==>{name=saturn, age=10000, type=titan}
==>{name=sky, type=location}
==>{name=sea, type=location}
==>{name=jupiter, age=5000, type=god}
==>{name=neptune, age=4500, type=god}
==>{name=hercules, age=30, type=demigod}
==>{name=alcmene, age=45, type=human}
==>{name=pluto, age=4000, type=god}
==>{name=nemean, type=monster}
==>{name=hydra, type=monster}
==>{name=cerberus, type=monster}
==>{name=tartarus, type=location}
gremlin>
```

GraphOfTheGodsFactory 类是 Titan 中自带用来建立和填入 Titan 数据库的实例图，它表现了罗马神话中的人物之间的关系和所处的地位。将目录路径传递给 create() 方法，就会返回一个 Blueprints 图的实现，明确了用一个 com.thinkaurelius.titan.graphdb.database.StandardTitanGraph 实例结合使用 BerkelyDB 和 Elasticsearch 作为后端存储。因为 Gremlin

shell 是一个 Groovy REPL，我们可以很容易看到 g 变量的类型：

```
gremlin> g.class.name
==>com.thinkaurelius.titan.graphdb.database.StandardTitanGraph
```

5.5　使用 Cassandra 存储后端设置 Titan

我们已经知道 Titan 支持多种后端存储。本章前面的部分已经分析了三种后端的优劣（可以在这里查找更详细的配置选项 http://thinkaurelius.github.io/titan/），所以我们最后选定使用 Cassandra（http://cassandra.apache.org/）存储后端。

5.5.1　安装 Cassandra

下载并解并运行 Cassandra，执行下述命令：

```
wget http://www.apache.org/dyn/closer.cgi?path=/cassandra/1.2.9/apache-
cassandra-1.2.9-bin.tar.gz
tar -zxf ./cassandra-1.2.9.bin.tar.gz
cd cassandra-1.2.9
./bin/cassandra -f
```

Cassandra 发行版的默认配置，会在本地建立一个单节点的数据库。如果安装报错，需要编辑 ${CASSANDRA_HOME}/conf/cassandra.yaml 和 ${CASSANDRA_HOME}/conf/log4j-server.properties 的配置文件。最常见的问题是程序没有 /var/lib/Cassandra（默认数据存储位置）和 /var/log/Cassandra（默认日志位置）的写权限。

5.5.2　使用 Cassandra 后端启动 Titan

为了让 Titan 启动时使用 Cassandra，需要修改配置连上 Cassandra 服务。建立一个新配置文件 storm-blueprints-cassandra.yaml 内容如下：

```
storage.backend=cassandra
storage.hostname=localhost
```

你可能已经猜出来了，这个配置连接到本地运行的实例。

注意　这个项目中，我们不需要真正运行 Titan 服务。因为我们使用 Cassandra，Storm 和 Gremlin 可以直接连接后端存储。

Titan 的后端配置好，我们开始建立数据模型。

5.6 图数据模型

我们的数据模型中最基本的实体是 Twitter 用户，一个 Twitter 用户发表推文时可以表现出以下关系形式：

- 用的词汇
- 提到话题
- 提到另外一个用户
- 提到一个 URL
- 转发其他用户

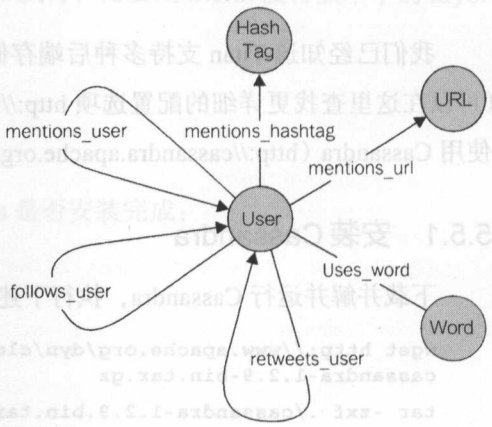


图 5-6

图 5-6 这个概念图是一个很自然的图形模型。在这个模型中，我们有四个不同的实体类型（顶点）：

- 用户（User）：代表 Twitter 用户帐号。
- 词汇（Word）：表示推文中包括的词汇。
- 链接（URL）：表示推文中包括的 URL 连接。
- 话题（Hashtag）：表示推文中提及的话题。

关系（边）包括以下动作：

- mentions_user：一个用户提及另一个用户。
- retweets_user：一个用户转发了另一个用户的消息。
- follows_user：一个用户关注了另一个用户。
- mentions_hashtag：一个用户提及了一个话题。
- uses_word：一个用户的推文中使用了特定词汇。
- mentions_url：一个用户的推文中包括特定 URL。

用户顶点刻画了一个 Twitter 用户的信息，如下表所示：

User[vertex]		
type	String	“user”
user	String	Twitter 账户名
name	String	Twitter 名称
location	String	Twitter 位置

URL 顶点表示指向唯一 URL 的引用：

URL[vertex]		
type	String	“url”
value	String	连接地址

话题顶点用来存储唯一的话题：

Hashtag[vertex]		
type	String	“hashtag”
value	String	

单词顶点存储单独的单词：

Word[vertex]		
type	String	“word”
value	String	

mentions_user 边用来表示用户之间的关系：

mentions_user[edge]		
user	String	提及的用户的 ID

mention_url 边用来表示用户和 URL 之间的关系：

mentions_url[edge]		
user	String	提及的用户的 ID

5.7　连接 Twitter 数据流

为了连接到 Twitter API，我们必须生成一些 OAuth token 用来做身份认证，让我们的程序能够连接到 Twitter。这个事情可以通过建立一个 Twitter 应用程序来实现，这个程序关联在 Twitter 账户上，然后可以使用这个程序访问账户。如果还没有 Twitter 账户，新建一个并登录。可以按照下列步骤生成 OAuth token：

1. 访问 https://dev.twitter.com/apps/new 并且登录账户。

2. 为你的应用输入描述说明。

3. 为应用输入一个 URL。在我们的例子中，这个 URL 不重要因为我们不是像手机应用一样建立发行版的应用。这里随意输入 URL 即可。

4. 提交表单，下一步会显示应用程序 OAuth 设置的详情。注意 Consumer key 和 Consumer secret 值是应用需要的。

5. 在页面底部，点击 Create my access token 按钮。这样会生成 OAuth 访问 token 和密钥，这样应用就可以使用这些访问账户。我们的应用中需要这些值，不要把这些内容告诉他人，否则别人会冒充你访问账户。

5.7.1 安装 Twitter4J 客户端

Twitter4J 客户端被分成了很多不同的模块，可以根据使用需求进行组合。在我们的例子中，需要使用到 core 模块提供基础功能比如 HTTP 传输、OAuth 和访问 Twitter API。我们还使用 stream 模块来访问数据流 API。这些模块可以被添加到 Maven 依赖中，如下代码所示：

```
<dependency>
  <groupId>org.twitter4j</groupId>
  <artifactId>twitter4j-core</artifactId>
  <version>3.0.3</version>
</dependency>
<dependency>
  <groupId>org.twitter4j</groupId>
  <artifactId>twitter4j-stream</artifactId>
  <version>3.0.3</version>
</dependency>
```

5.7.2 OAuth 配置

默认情况下，Twitter4J 会检索 classpath 来查找 twitter4j.properties 文件，并且从文件中载入 OAuth token。最简单的方法是在 Maven 项目的 resources 文件中建立下列文件：

```
oauth.consumerKey=[your consumer key]
oauth.consumerSecret=[your consumer secret]
oauth.accessToken=[your access token]
oauth.accessTokenSecret=[your access token secret]
```

我们现在可以使用 Twitter4J 客户端来连接 Twitter 的数据流 API，并从中实时读取推文消息了。

5.7.3 TwitterStreamConsumer 类

我们使用 Twitter 客户端的目的很明确，用到了下列功能：

- 连接到 Twitter 数据流 API
- 使用一组关键字过滤推文的数据流
- 将推文封装为 JSON 数据结构
- 将 JSON 数据写入到 Kafka 供 Kafka spout 来消费

TwitterStreamConsumer 类中的 main() 方法新建了一个 TwitterStream 对象，并且注册了一个 StatusListener 实例作为监听器。StatusListener 接口用做异步事件 handler，当 stream 相关的事件发生时会被通知：

```
    public static void main(String[] args) throws TwitterException,
IOException {

        StatusListener listener = new TwitterStatusListener();
        TwitterStream twitterStream = new TwitterStreamFactory().
getInstance();
        twitterStream.addListener(listener);

        FilterQuery query = new FilterQuery().track(args);
        twitterStream.filter(query);
    }
```

注册了监听器之后，建立一个 FilterQuery 对象来按照关键字过滤数据流。为简化起见，我们使用程序的参数输入作为过滤的关键字列表，这样过滤条件就可以根据命令行参数进行变换。

5.7.4　TwitterStatusListener 类

TwitterStatusListener 类在我们应用中执行了大部分负载。StatusListener 类定义了多个回调函数作用于 stream 的生命周期中出现的事件。onStatus() 方法是我们最关心的，因为每来一个新推文时该方法都会调用一次。下面就是 TwitterStatusListener 类：

```
    public static class TwitterStatusListener implements
StatusListener {
        public void onStatus(Status status) {

            JSONObject tweet = new JSONObject();
            tweet.put("user", status.getUser().getScreenName());
            tweet.put("name", status.getUser().getName());
            tweet.put("location", status.getUser().getLocation());
            tweet.put("text", status.getText());
```

```
                HashtagEntity[] hashTags = status.getHashtagEntities();
                System.out.println("# HASH TAGS #");
                JSONArray jsonHashTags = new JSONArray();
                for (HashtagEntity hashTag : hashTags) {
                    System.out.println(hashTag.getText());
                    jsonHashTags.add(hashTag.getText());
                }
                tweet.put("hashtags", jsonHashTags);

                System.out.println("@ USER MENTIONS @");
                UserMentionEntity[] mentions = status.
getUserMentionEntities();
                JSONArray jsonMentions = new JSONArray();
                for (UserMentionEntity mention : mentions) {
                    System.out.println(mention.getScreenName());
                    jsonMentions.add(mention.getScreenName());
                }
                tweet.put("mentions", jsonMentions);

                URLEntity[] urls = status.getURLEntities();
                System.out.println("$ URLS $");
                JSONArray jsonUrls = new JSONArray();
                for (URLEntity url : urls) {
                    System.out.println(url.getExpandedURL());
                    jsonUrls.add(url.getExpandedURL());
                }
                tweet.put("urls", jsonUrls);

                if (status.isRetweet()) {
                    JSONObject retweetUser = new JSONObject();
                    retweetUser.put("user", status.getUser().
getScreenName());
                    retweetUser.put("name", status.getUser().getName());
                    retweetUser.put("location", status.getUser().
getLocation());
                    tweet.put("retweetuser", retweetUser);
                }
                KAFKA_LOG.info(tweet.toJSONString());
            }

            public void onDeletionNotice(StatusDeletionNotice
statusDeletionNotice) {
            }

            public void onTrackLimitationNotice(int
numberOfLimitedStatuses) {
                System.out.println("Track Limitation Notice: " +
```

```
numberOfLimitedStatuses);
        }

        public void onException(Exception ex) {
            ex.printStackTrace();
        }

        public void onScrubGeo(long arg0, long arg1) {
        }

        public void onStallWarning(StallWarning arg0) {

        }
    }
```

在状态消息的原始内容之外，Status 对象包括了很方便的方法来访问关联的元数据，比如用户信息以及推文中提到的话题、URL 和户。onStatus() 方法中的代码建立了一个 JSON 结构，它被 Logback Kafka Appender 写进 Kafka 队列。

5.8　Twitter graph topology

Twitter graph topology 会从 kafka 队列中读取推特消息数据，解析出相关信息，然后在 Titan 图数据库中建立节点和关系。并不是每个 tuple 到来时都操作图数据库一次，我们会使用 Trident 的传输机制，开发一个 Trident state 实现，将数据成批进行存储。

这种实现方式有多种好处。首先，Titan 这类图形数据库支持事务操作，我们可以利用这个特性来实现额外的数据处理唯一性保证。其次，这样实现允许通过使用批量提交（当功能支持的时候）将一组 tuple 一次性批量提交而不是一个个提交。最后，使用通用的 Blueprints API，我们的 Trident state 实现可以忽略底层图数据库的实现，任何支持 Blueprints 的图形数据库后端都可以方便地切换，如图 5-7 所示。

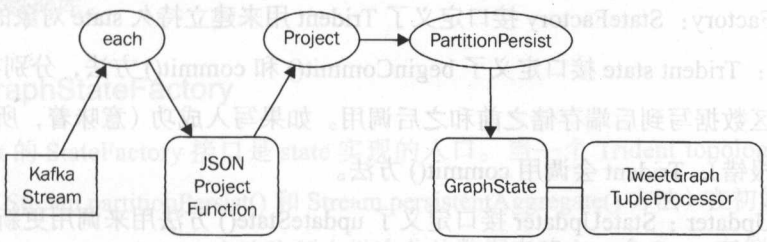


图　5-7

topology 的第一个组件包括了 JSONProjectFunction，我们在第 7 章中开发的，这个方法会解析原始 JSON 数据提取我们关心的部分。在这个例子中，我们主要关心消息的时间戳和 JSON 中表示的 Twitter 状态消息。

JSONProjectFunction 类

下列代码片段解释了 JSONProjectFunction 类：

```
public class JsonProjectFunction extends BaseFunction {

    private Fields fields;

    public JsonProjectFunction(Fields fields) {
        this.fields = fields;
    }

    public void execute(TridentTuple tuple, TridentCollector
collector) {
        String json = tuple.getString(0);
        Map<String, Object> map = (Map<String, Object>) JSONValue.
parse(json);
        Values values = new Values();
        for (int i = 0; i < this.fields.size(); i++) {
            values.add(map.get(this.fields.get(i)));
        }
        collector.emit(values);
    }

}
```

5.9 实现 GraphState

topology 的核心是一个 Trident state 的实现用来将 Trident 的 tuple 转换为图数据结构并且持久化存储。回顾一下，一个 Trident state 实现包括下面三个组件：

- StateFactory：StateFactory 接口定义了 Trident 用来建立持久 state 对象的方法。
- State：Trident state 接口定义了 beginCommit() 和 commit() 方法，分别在 Trident 一批分区数据写到后端存储之前和之后调用。如果写入成功（意味着，所有的处理都没有报错），Trident 会调用 commit() 方法。
- StateUpdater：StateUpdater 接口定义了 updateState() 方法用来调用更新 state，假定处理了一批 tuple。Trident 将三个参数传递给这个方法，需要更新的 state 对象，一

个批次分区数据中 TridentTuple 对象的列表，和 TridentCollector 实例用来视需要发送额外的 tuple 作为 state 更新的结果。

在 Trident 提供的这些抽象之外，我们还介绍两个额外的接口，用来和 Blueprints 图数据库（GraphFactory）交互以及用来隔离实际的业务处理逻辑（GraphTupleProcessor）。在开始实现 Trident state 之前，先快速了解下这些接口。

5.9.1　GraphFactory

GraphFactory 接口非常简单：提供一个表示 Storm 和 topology 配置信息的 Map 对象，返回一个 com.tinkerpop.blueprints.Graph 实现。

```
GraphFactory.java
public interface GraphFactory {
    public Graph makeGraph(Map conf);
}
```

通过实现 makeGraph 方法，这个接口允许我们使用任何兼容 Blueprints 接口的图实现。后面我们会实现这个接口，返回一个到 Titan 图数据库的连接。

5.9.2　GraphTupleProcessor

GraphTupleProcessor 接口提供了一个在 Trident state 实现和业务逻辑之间的抽象。

```
public interface GraphTupleProcessor {

    public void process(Graph g, TridentTuple tuple, TridentCollector
collector);

}
```

GraphTupleProcessor 的主要工作是，对给定的图形对象，TridentTuple 和 TridentCollector，操作图形对象并且选择性的发送额外的 tuple。本章后面会实现这个接口，将 Twitter 状态消息填入图数据库。

5.9.3　GraphStateFactory

Trident 的 StateFactory 接口是 state 实现的入口。当一个 Trident topology 使用的 state 组件（通过 Stream.partitionPersist() 和 Stream.persistentAggregate() 方法）在初始化时，Storm 调用 StateFactory.makeState() 方法为每个批次分片数据来建立一个 State 实例。批次分片的

个数取决于 steam 的并发量。Storm 将信息通过 numPartitions 和 partitionIndex 参数传递给 makeState() 方法，使 state 实现在需要时可以进行针对分区的特殊逻辑。在我们的例子里，我们不关心分片情况，所以 makeState() 方法仅使用 GraphFactory 实例来实现一个 Graph 实例用来构造 GraphState 实例。

```
GraphStateFactory.java
public class GraphStateFactory implements StateFactory {

    private GraphFactory factory;

    public GraphStateFactory(GraphFactory factory){
        this.factory = factory;
    }
    public State makeState(Map conf, IMetricsContext metrics, int
partitionIndex, int numPartitions) {
        Graph graph = this.factory.makeGraph(conf);
        State state = new GraphState(graph);
        return state;
    }

}
```

5.9.4 GraphState

我们的 GraphState 类实现了 State.beginCommit() 和 State.commit() 方法，会在一个批分片数据将要产生和完成时调用。我们的例子中，重写了 commit() 方法来检查内部的 Graph 对象是否支持事务，如果支持，则调用 TransactionalGraph.commit() 来完成事务。

如果 Trident 一批数据中出现处理错误或者一批数据被重放，State.beginCommit() 可能会被调用多次，State.commit() 方法则知会在所有分片状态成功更新时调用一次。

GraphState 类的代码段如下：

```
GraphState.java
public class GraphState implements State {

    private Graph graph;

    public GraphState(Graph graph){
        this.graph = graph;
    }
```

```
    @Override
    public void beginCommit(Long txid) {}

    @Override
    public void commit(Long txid) {
        if(this.graph instanceof TransactionalGraph){
            ((TransactionalGraph)this.graph).commit();
        }
    }
    public void update(List<TridentTuple> tuples, TridentCollector
collector, GraphTupleProcessor processor){
        for(TridentTuple tuple : tuples){
            processor.process(this.graph, tuple, collector);
        }
    }

}
```

5.9.5　GraphUpdater

GraphUpdater 类实现了 updateState() 方法，Storm 在调用 State.beginCommit() 之后调用这个方法（处理失败或者数据重放会导致重复调用）。StateUpdater.updateState() 方法的第一个参数是一个 state 实现的 Java 泛型实例，我们用来调用 GraphState.update() 方法。

```
GraphUpdater.java
public class GraphUpdater extends BaseStateUpdater<GraphState> {

    private GraphTupleProcessor processor;

    public GraphUpdater(GraphTupleProcessor processor){
        this.processor = processor;
    }

    public void updateState(GraphState state, List<TridentTuple>
tuples, TridentCollector collector) {
        state.update(tuples, collector, this.processor);
    }

}
```

5.10　实现 GraphFactory

我们前面定义的 GraphFactory 接口建立了一个 ThinkerPop 图的实现，Map 对象存储

了 Storm 的配置信息。下面的代码示例说明如何建立一个后端是 Cassandra 的 TitanGraph 图：

```
TitanGraphFactory.java
public class TitanGraphFactory implements GraphFactory {

    public static final String STORAGE_BACKEND = "titan.storage.
backend";
    public static final String STORAGE_HOSTNAME = "titan.storage.
hostname";

    public Graph makeGraph(Map conf) {
        Configuration graphConf = new BaseConfiguration();
        graphConf.setProperty("storage.backend", conf.get(STORAGE_
BACKEND));
        graphConf.setProperty("storage.hostname", conf.get(STORAGE_
HOSTNAME));

        return TitanFactory.open(graphConf);
    }
}
```

5.11 实现 GraphTupleProcessor

为了将 Twitter 状态信息中搜集到的关系填入图形数据库，我们需要实现 GraphTupleProcessor 接口。下述代码示例如何解析 Twitter 状态信息的 JSON 对象，并且建立“user”和“hashtag”顶点之间的关系“mentions”。

```
TweetGraphTupleProcessor.java
public class TweetGraphTupleProcessor implements GraphTupleProcessor {
    @Override
    public void process(Graph g, TridentTuple tuple, TridentCollector
collector) {
        Long timestamp = tuple.getLong(0);
        JSONObject json = (JSONObject)tuple.get(1);

        Vertex user = findOrCreateUser(g, (String)json.get("user"),
(String)json.get("name"));
        JSONArray hashtags = (JSONArray)json.get("hashtags");
        for(int i = 0; i < hashtags.size(); i++){
            Vertex v = findOrCreateVertex(g, "hashtag", ((String)
hashtags.get(i)).toLowerCase());
            createEdgeAtTime(g, user, v, "mentions", timestamp);
        }

    }
}
```

5.12　组合成 TwitterGraphTopology 类

通过下面几步就可以建立最终的 topology：

- 从 Kafka spout 中消费原始 JSON 数据。
- 提取和投射我们关心的数据。
- 建立 Trident GraphState 实例并连接到数据流上。

TwitterGraphTopology 类

让我们看看 TwitterGraphTopology 类的详细实现：

```
public class TwitterGraphTopology {
    public static StormTopology buildTopology() {
        TridentTopology topology = new TridentTopology();

        StaticHosts kafkaHosts = StaticHosts.fromHostString(Arrays.
asList(new String[] { "localhost" }), 1);
        TridentKafkaConfig spoutConf = new
TridentKafkaConfig(kafkaHosts, "twitter-feed");
        spoutConf.scheme = new StringScheme();
        spoutConf.forceStartOffsetTime(-2);
        OpaqueTridentKafkaSpout spout = new OpaqueTridentKafkaSpout(s
poutConf);

        Stream spoutStream = topology.newStream("kafka-stream",
spout);

        Fields jsonFields = new Fields("timestamp", "message");
        Stream parsedStream = spoutStream.each(spoutStream.
getOutputFields(), new JsonProjectFunction(jsonFields), jsonFields);
        parsedStream = parsedStream.project(jsonFields);
        // Trident State
        GraphFactory graphFactory = new TitanGraphFactory();
        GraphUpdater graphUpdater = new GraphUpdater(new
TweetGraphTupleProcessor());

        StateFactory stateFactory = new GraphStateFactory(graphFacto
ry);
        parsedStream.partitionPersist(stateFactory, parsedStream.
getOutputFields(), graphUpdater, new Fields());

        return topology.build();
    }

    public static void main(String[] args) throws Exception {
```

```
        Config conf = new Config();
        conf.put(TitanGraphFactory.STORAGE_BACKEND, "cassandra");
        conf.put(TitanGraphFactory.STORAGE_HOSTNAME, "localhost");

        conf.setMaxSpoutPending(5);
        if (args.length == 0) {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("twitter-analysis", conf,
buildTopology());

        } else {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopology(args[0], conf,
buildTopology());
        }
    }
}
```

要运行这个应用，首先要执行 TwitterStreamConsumer 类，将关键字列表传递给 Twitter 信息源。例如，如果你想建立一个讨论大数据的用户的图，可以使用 bigdata 和 hadoop 作为查询关键字：

```
java TwitterStreamConsumer bigdata hadoop
```

TwitterStreamConsumer 类会连接到 Twitter 流 API，并且开始将数据发送到 Kafka 队列。当 TwitterStreamConsumer 程序运行时，我们就可以将 TwitterGraphTopology 发布开始将数据填入 Titan 数据库。

让 TwitterStreamConsumer 和 TwitterGraphTopology 执行一会儿。取决于查询关键词的热度，可能需要等一会数据量才能够累积到足够的级别。然后我们就可以使用 Gremlin shell 连接 Titan 数据库来使用图查询功能进行分析。

5.13 使用 Gremlin 查询图

为了查询图我们需要运行 Gremlin shell，建立一个 TitanGraph 实例连接到本地的 Cassandra 后端：

```
$ cd titan
$ ./bin/gremlin.sh
          \,,,/
         (o o)
-----oOOo-(_)-oOOo-----
```

```
gremlin> conf = new BaseConfiguration()
gremlin> conf.setProperty('storage.backend', 'cassandra')
gremlin> conf.setProperty('storage.hostname', 'localhost')
gremlin> g = TitanFactory.open(conf)
```

g 变量包括了一个 Graph 的对象，我们可以用来进行图遍历查询。下面有一些查询示例：

- 为了查找所有发表了 #hadoop 话题的用户，并且显示他们发表这些话题的次数，使用下列代码：

```
gremlin> g.V('type', 'hashtag').has('value', 'hadoop').in.userid.
groupCount.cap
```

- 为了计算 #hadoop 话题被发表推文的次数，使用下列代码：

```
gremlin> g.V.has('type', 'hashtag').has('value', 'java').inE.
count()
```

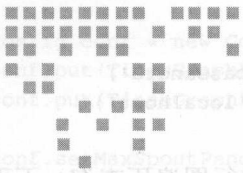
Gremlin DSL 非常强大，包括了完整的 API 可以满足整章的需求（如果不是全书的话）。如果要进一步了解 Gremlin 语言，我们建议你去研究下列在线文档：

- 官方 Wiki https://github.com/tinkerpop/gremlin/wiki。
- GremlinDocs 参考指南 http://gremlindocs.com。
- SQL2Gremlin（SQL 语句示例及其 Gremlin 的实现）http://sql2gremlin.com。

总结

在本章中，我们介绍了图形数据库，建立一个 topology 来监控 Twitter 信息源，将信息持久化存储在 Titan 图数据库以便进行进一步分析。我们还通过使用前面章节的通用代码，比如 Logback Kafka appender，来举例说明如何复用组件。

虽然图形数据库不是全能的，但它确实是一种适用于混合持久化（Polyglot Persistence）的强力工具。混合持久化这个术语表示这样一种软件架构。包括多种数据存储类型，如关系、key-value、图形、文件等。混合持久化说白了就是选择合适的数据库去干合适的事情。在本章中，我们介绍了图数据模型，建议读者继续研究在何种情况下最适合使用图数据模型。在本书后面，我们会建立一个 Storm 应用，为了不同的目的，将数据持久化写入不同类型的数据库中。



Chapter 6 第6章

人工智能

在前面的章节中，我们了解有一种模式结合了 Storm 的实时分析和 Hadoop 的批处理特性。本章我们将换个方向。我们将 Storm 整合到一个运营系统中，这个系统必须实时反馈用户的查询。

Storm 典型的应用集中在无尽的数据流上。这些数据通常在 topology 保持稳定运行的前提下尽可能快地进入队列并处理。系统包含了一个可以自适应条件大负载流量的队列。在负载较低时，队列是空闲的。在负载高时，队列会将数据持久化到硬盘上等待事件的进一步处理。

即使是不懂行的人也能看出，这样的系统不能提供真正的实时数据处理。Storm 会监控 tuple 的超时，但是它主要关注 spout 发射数据后处理 tuple 的时长。

为了更好地支持实时场景，超时和服务层确认（Service Level Agreements，SLA）必须从数据接收起一直监控到返回应答为止。目前，请求通常是基于 HttP 的 API 并且响应时间要达到亚秒级。

HTTP 是一个同步协议。它常常还会引入一个异步机制比请求队列，这使系统变得更加复杂，也增加了系统延迟。因此，使用 HTTP 特性或者功能时，我们通常倾向于在涉及的组件中使用同步机制。

本章中，我们介绍 Storm 在 Web 服务 API 架构中所处的位置。特别是，我们会建立世界上最好的井字游戏人工智能（Artificial Intelligence，AI）系统。这个系统会包括同步

和异步的子系统，系统中的异步部分会持续运行，计算游戏中最优选择。同步部分组件会提供一个 Web 服务接口，给出一个游戏状态，返回最佳的移动线路。

本章包括下列主题：

- 在 Storm 中实现回归。
- 分布式远程命令调用（DRPC）。
- 分布式系统写入前读取数据范例。

6.1 为应用场景进行设计

人工智能世界中，最入门的例子是井字棋游戏。按照惯例，我们会使用这个游戏作为主题进行游戏，但它的体系结构和实现方法的扩展性远在这个简单的例子之上（比如，另外一个例子是 John Badham War Games 中的全球热核战争）。

井字棋游戏是两个玩家使用打圈（O）和打叉（×）进行的游戏。在一个 3×3 的格子里。一个用户使用符号 O，另一个使用符号 ×，轮流画记号。每轮玩家将自己的符号画在空白处，首先将三个相同符号完成水平，垂直或者斜线连接的人取胜，游戏结束。

为这种轮番步骤的游戏，人工智能程序的通用实现方式是树形递归查找当前游戏玩家的最优解（或者最坏选择）。游戏树是一个树形结构，每个顶点是一个游戏状态。一个节点的直接子节点是当前游戏节点代表的游戏状态通过合法步骤所能走出的所有步骤。

井字棋游戏一个示例的游戏树如图 6-1 所示。

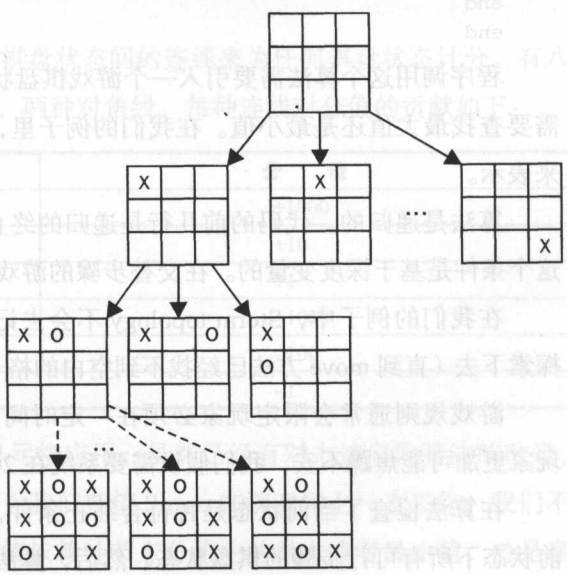


图 6-1

遍历一个游戏树查找最佳步骤的最简单的算法之一是极小极大算法 Minimax algorithm。这个算法采取遍历的方法给每个棋盘状态算出最佳分数。这个算法中，我们假设一个对手的好的分数就是当前玩家的坏分数。这样的话，

算法实际上就是求每个当前棋盘状态的最大值和最小值。极小化极大算法可以总结为下列伪代码：

```
miniMax (board, depth, maximizing)
   if (depth <= 0)
      return score (board)
   else
      children = move(board)
      if (maximizing)
         bestValue = -∞
      for (child : children)
         value = miniMax (child, depth-1, false)
         if (value > bestValue)
            bestValue = value
         end
end
return bestValue
      else // minimizing
         bestValue = ∞
      for (child : children)
         value = miniMax (child, depth-1, false)
         if (value < bestValue)
            bestValue = value
         end
end
return bestValue
end
end
```

程序调用这个算法需要引入一个游戏棋盘状态，一个深度，一个布尔值表示当前算法需要查找最大值还是最小值。在我们的例子里，游戏状态通过在 3 × 3 的格子里填 O 和 X 来表示。

算法是递归的。代码的前几行是递归的终止条件。这保证了算法不会出现无限递归。这个条件是基于深度变量的。在交替步骤的游戏中，深度决定了算法会探索多少步骤。

在我们的例子中，Storm topology 不会去记录深度。我们会让 Storm topology 无线的探索下去（直到 move 方法已经找不到空白的格子）。

游戏规则通常会限定玩家必须在一定时间内走出下一步。因为在和 AI 进行游戏时，玩家更加可能焦躁不安，我们假设需要系统在 200 毫秒内给出响应。

在算法检查了当前状态是否符合终止条件，会调用 move() 方法，这个方法会返回当前状态下所有可行步骤的棋盘状态。然后，算法会循环所有可能的子节点的棋盘状态。如果要最大化值，算法会算出当前子节点中分值最高的一个格子，如果要最小化值，算法会

选择最小值分的棋盘状态。

负极大值算法通过交替变换分值的正负号，更简洁地实现了同样的功能。此外，在实际生活的场景中，我们可能会应用 Alpha–beta pruning 算法，用来尝试对探索的游戏树进行剪枝。算法只考虑落在阈值之内的分支。在我们的例子里，没有必要使用这种算法，因为整体的搜索空间非常小。

在我们简单的使用场景中，枚举整个游戏树是可能的。在另一些复杂的游戏，比如国际象棋中，游戏树的规模太大导致无法完全枚举。在极端的场景下如围棋，专家计算出合法的下法超过了 2 的 10170 次方。

极小极大算法的目标是遍历游戏树，赋给每个节点一个分值。我们 Storm topology 并不遵循任何服务水平协议（SLA），每个非叶子节点的分值就是它后代的最小或者最大值。对于一个叶子节点，我们必须将游戏状态解释为一个对应的分值。在我们的例子中，有三种可能的结果，我们赢，对手赢，或者平局。

但是在我们的同步系统中，我们可能在达到叶节点之前就超过时间限制。在这种场景下，我们需要计算棋盘当前状态的分值。计分的启发算法往往是开发人工智能程序中最难的部分。

在我们简单的示例中，我们通过计算棋盘状态间的连线来为任何棋盘状态计分。有八种需要考虑的连线。三种水平，三种垂直，两种对角线。每种连线对分值的贡献如下：

状　态	分　值
当前玩家三个格子连成线	+1000
当前玩家两个格子连成线	+10
当前玩家一个格子	+1
对手三个格子连成线	−1000
对手两个格子连成线	−100
对手一个格子	−1

上面的表格只对每条连线中空白的单元格应用。虽然已经有对上述启发算法的改进，但上面的算法对这个例子已经够用了。因为我们期望 Storm 在游戏树上一直工作，我们不希望太多的依赖启发算法。相反，我们直接依赖叶节点分值中的最大或者最小值，总是赢（+1000），输（−1000）或者平（0）。

我们现在已经有了实现方法，算法和计分函数，我们可以继续介绍体系结构和设计了。

6.2 确立体系结构

回想前面的算法，这里有很多有趣的设计和体系结构上的考虑，特别是在运用 Storm 的情况下。算法需要递归。我们还需要一种同步处理请求的方式。在 Storm 中进行递归操作仍是一个正在讨论中待完善的主题，同时 Storm 已经提供了一种和 topology 同步进行交互的方法，将递归的需求结合进来之后，为我们的设计形成了一种特别和有趣的挑战。

6.2.1 审视设计中的挑战

最初，原生的 Storm 提供了一种异步命令调用的服务机制。这个特性是分布式远程命令调用（Distributed Remote Procedure Call，DRPC）。DRPC 允许客户端向 topology 提交数据发起请求。在 DRPC 中，一个 RPC 客户端扮演了一个 spout 的角色。

Trident 出现后，DRPC 在原生 Storm 中已经不建议使用了，当前官方只在 Trident 中支持 DRPC 功能。

尽管对递归 / 非线性 DPRC 已经有一些探索性的工作，就是我们这里需要的功能，现在还不是主干功能（https://groups.google.com/forum/#!topic/storm-user/hk3opTiv3Kc）。

另外，上面的工作依赖与原生 Storm 中一些已经废弃的类。因此我们需要寻找替代方法来建立不依赖与原生 Storm 的递归结构。

一旦我们实现了递归结构，会采用同步的方式调用这个功能。尝试复用 Storm 已经提供的组件，意味着要把 DRPC 调用集成到我们的体系中。

6.2.2 实现递归

如果我们将算法映射到 Storm 结构中，我们可能期望一个方法允许数据流将结果数据反馈给它自身。我们可以想想下类似图 6-2 中逻辑数据流的 topology。

BoardSpout 函数在 currentBoard 字段中发射一个棋盘状态（例如，3 × 3 的数组），第二个叫做 parents 的字段用来存储他所有的祖先节点。Parents 节点在最开始时是空的。

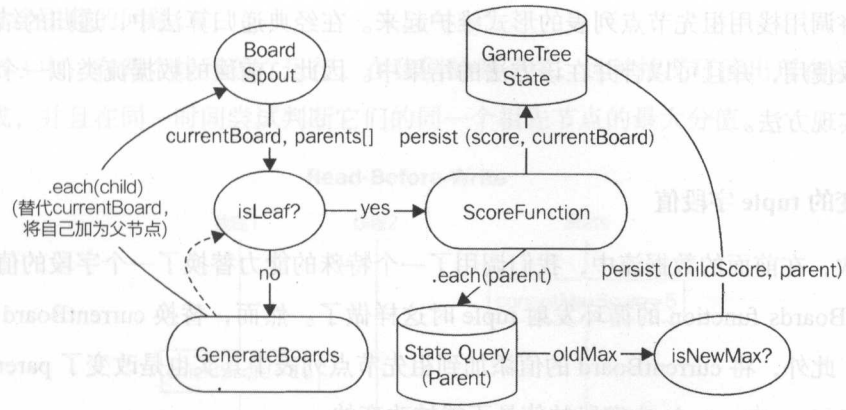


图 6-2

过滤器 isLeaf 决定是不是已经到达了终止状态（例如，赢，输，平局）。如果 currentBoard 的值不是结束状态，GenerateBoards function 发射所有新的棋盘子状态，使用子节点棋盘状态的值代替 currentBoard 的值，并且将当前 currentBoard 的值添加到 parents 字段中的节点列表中。GenerateBoards function 可以将 tuple 发射回给 spout 或者绕过 spout 直接发射给 isLeaf filter。

如果 isLeaf filter 判定当前是结束状态，我们需要给 currentBoard 字段中的棋盘状态打分，然后用新的分数更新所有的祖先节点。ScoreFunction 计算棋盘状态的分值，并且持久化到 GameTree State 中。

为了更新父节点，我们遍历所有的祖先节点并且查询当前节点的最大（或最小）值。如果子节点的分值是新的最大（或最小）值，我们就会持久化新的值。


提示 这只是一个逻辑数据流，构造这样的 topology 不但是不可行的，而且也是不建议的，原因会在后续章节中描述。

你已经看到了，这个数据流不像伪代码那样简单明了。Trident 和 Storm 的一些约束条件迫使我们引入了额外的复杂度，而且，数据流的操作链中不是所有操作都是 Storm/Trident 支持的。让我们近距离的审视这个数据流。

访问 function 的返回值

首先，因为 Storm 和 Trident 在 topology 中不提供访问下游 function 结果的机制，我

们必须将调用栈用祖先节点列表的形式维护起来。在经典递归算法中，递归的结果可以直接被函数使用，并且可以合并在该方法的结果中。因此，前面的数据流类似一个多次迭代的递归实现方法。

不变的 tuple 字段值

其次，在前面的数据流中，我们调用了一个特殊的能力替换了一个字段的值。我们在 GenerateBoards function 的循环发射 tuple 时这样做了。然而，替换 currentBoard 的值是不可能的。此外，将 currentBoard 的值添加到祖先节点列表中其实也是改变了 parents 字段的值。在 Trident 中，tuple 中字段的值是不能被改变的。

预先的字段声明

为了避开 tuple 值的不变性，我们可以总是给 tuple 添加新的字段，每层递归添加一个字段，但是 Trident 需要所有的字段在 toplogy 发布前就必须声明好。

递归中的 tuple 确认应答

考虑这个数据流中 tuple 的应答确认时，又遇到一个新的问题。什么时候应该对触发了该次数据处理的 tuple 进行确认应答？从逻辑数据流的角度来看，最初的 tuple 在所有的子节点都被考虑完并且给游戏树给出了对应分值之前是不应该确认应答。然而可以确定的是，计算稍复杂游戏的游戏树很可能超过任何 tuple 设置的超时时间。

向多个 stream 输出

Toplogy 的另外一个问题就是 isLeaf filter 发送了多个输出路径。目前在 Trident 中还没有发射多个 stream 的方法。这个等待增加的功能可以在 https://issues.apache.org/jira/browse/STORM-68 找到描述。

为了绕过这个限制，可以通过将数据流进行分支，在每个分支数据流上都应用 filter。

写入之前读取

最后，因为我们无法访问到返回值，更新祖先节点的分值需要一个写入之前读取的范式。在所有分布式系统中这都一个的反面模式。图 6-3 示例了在没有锁机制下，写入之前

读取设计会引起的问题。

图 6-3 中，有两个线程独立运行。在我们的例子中，这种情况可能出现在多个子节点同时完成，并且在同一时间尝试判断它们的同一个祖先节点的最大分值。

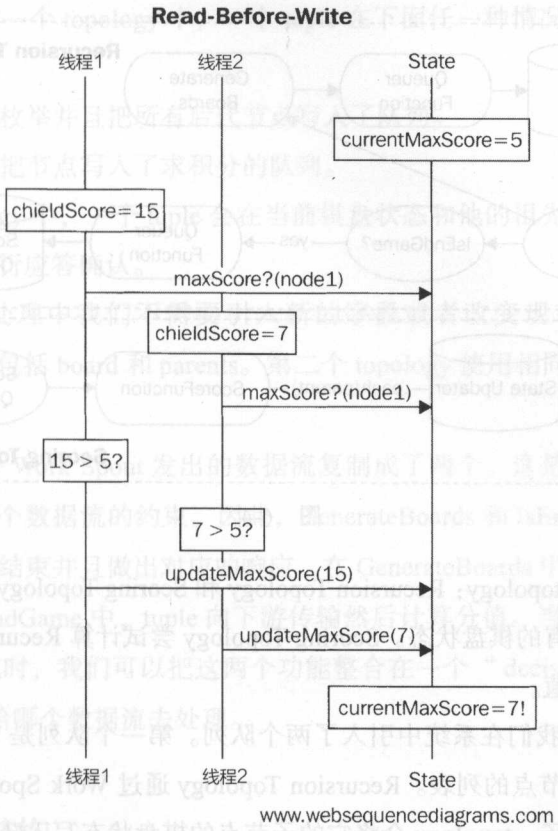


图 6-3

第一个线程求出一个子节点的分值是 7，第二个线程求出的分值是 15。它们同时去计算一个祖先节点的值。处理完成时，最新的最大值应该是 15，但是因为两个线程直接没有同步机制，最后最大值是 7。

第一个线程获取当前的最大分值，返回 5. 然后第二个线程查询状态也收到 5。两个线程都必将当前的最大值和它们各自子节点求出的值，并且更新新的最大值。因为第二个线程更新操作比较晚，最后父节点的值被更新为错误的值。

下一节中，我们会看到如何解决这些制约，实现这个功能的系统。

6.2.3 解决这些挑战

为了适应上一节中提到的限制条件，我们将会讲 topology 分为两个部分：第一个部分执行实际的递归操作，第二个部分执行计算分值，如图 6-4 所示。

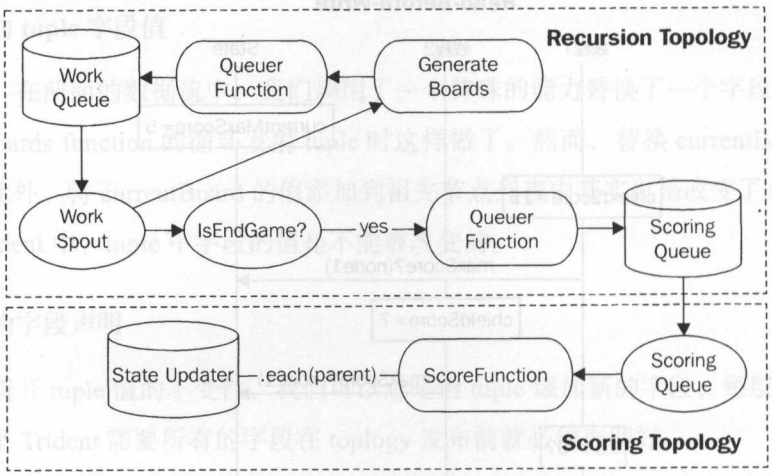


图 6-4

系统划分为两个 topology：Recursion Topology 和 Scoring Topology。Recursion Topology 尝试在系统中枚举所有的棋盘状态。Scoring Topology 尝试计算 Recursion Topology 枚举出的所有棋盘状态的分值。

为了引起递归，我们在系统中引入了两个队列。第一个队列是 Work Queue，包含了一个我们需要访问的节点的列表。Recursion Topology 通过 Work Spout 从该队列中消费数据。如果节点不是叶子，topology 会将它的子节点的棋盘状态写入队列。队列中消息的格式如下：

```
(board, parents[])
```

每个 board 是一个 3X3 的数组，parent 数组包括了所有的祖先节点的棋盘状态。

如果节点是一个叶节点，会将棋盘状态用相同的消息格式放入 Scoring Queue。Scoring Topology 使用 Scoring Spout 从 Scoring Queue 中读取消息。Scoring Function 计算节点的分值。这个队列中的节点都是叶节点，因为只有叶子节点才会写入这个队列计算分值。然后，Scoring Function 发射当前节点的分值以及这个节点的祖先节点。

我们然后需要更新状态信息。因为可能出现前面提到的竞争状态，我们将查询和写的

范式被封装在了一个单独的 function 中。在下面的设计中，我们将会讲到如何避免写入之前读取引入的竞争状态。

在我们开始介绍设计之前，要注意到因为我们引入了队列，我们要明确如何对 tuples 进行确认应答。在第一个 topology 中，一个 tuple 在下面任一种情况为真的情况下进行确认应答：

- topology 已经枚举并且把所有后代节点写入了队列。
- topology 已经把节点写入了求积分的队列。

在第二个 topology 中，一个 tuple 会在当前棋盘状态和他的祖先都已经根据叶节点的分值进行更新之后进行应答确认。

还要注意，在处理中我们不需要引入新的字段或者改变现有字段的值。第一个 topology 使用的字段包括 board 和 parents。第二个 topology 使用相同的字段，然后增加对应的分值字段。

还要注意我们将 Work Spout 发出的数据流复制成了两个。这是为了绕过不能从同一个 function 中发射多个数据流的约束。因此，GenerateBoards 和 IsEndGame 两个组件分别都需要决定游戏是否结束并且做出对应的响应。在 GenerateBoards 中，tuple 会被过滤以避免无限递归。在 IsEndGame 中，tuple 向下游传输然后计算分值。当 functions 将来支持可以发射不同的数据流时，我们可以把这两个功能整合在一个“decision”过滤器中，决定一个 tuple 应该发射给哪个数据流去处理。

6.3 实现体系结构

让我们开始研究实现的细节。作为示例的目的，下面代码假设 topology 运行在本地环境。我们使用一个内存队列代替持久化队列，使用一个 hashmap 作为我们的存储机制。在生产环境的实现中，我们更倾向于使用一个持久化队列，比如 Kafka，一个分布式存储系统，比如 Cassandra。

6.3.1 数据模型

我们将深入了解这两个 topology，但在这之前，先要了解一下数据模型。为了简化起见，我们将游戏逻辑和数据模型封装成了两个类：Board 和 GameState。

下面的代码是棋盘状态 Board 类：

```
public class Board implements Serializable {
public static final String EMPTY = ' ';
   public String[][] board = { { EMPTY, EMPTY, EMPTY },
{ EMPTY, EMPTY, EMPTY }, { EMPTY, EMPTY, EMPTY } };

public List<Board> nextBoards(String player) {
        List<Board> boards = new ArrayList<Board>();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j].equals(EMPTY)) {
                    Board newBoard = this.clone();
                    newBoard.board[i][j] = player;
                    boards.add(newBoard);
                }
            }
        }
        return boards;
    }

    public boolean isEndState() {
        return (nextBoards('X').size() == 0
|| Math.abs(score('X')) > 1000);
    }

    public int score(String player){
        return scoreLines(player) -
            scoreLines(Player.next(player));
    }
    public int scoreLines(String player) {
        int score = 0;
        // Columns
        score += scoreLine(board[0][0], board[1][0], board[2][0],
player);
        score += scoreLine(board[0][1], board[1][1], board[2][1],
player);
        score += scoreLine(board[0][2], board[1][2], board[2][2],
player);

        // Rows
        score += scoreLine(board[0][0], board[0][1], board[0][2],
player);
        score += scoreLine(board[1][0], board[1][1], board[1][2],
player);
        score += scoreLine(board[2][0], board[2][1], board[2][2],
player);
```

```
        // Diagonals
        score += scoreLine(board[0][0], board[1][1], board[2][2],
player);
        score += scoreLine(board[2][0], board[1][1], board[0][2],
player);
        return score;
    }

    public int scoreLine(String pos1, String pos2, String pos3, String
player) {
        int score = 0;
        if (pos1.equals(player) && pos2.equals(player) && pos3.
equals(player)) {
            score = 10000;
        } else if ((pos1.equals(player) && pos2.equals(player) &&
pos3.equals(EMPTY)) ||
                (pos1.equals(EMPTY) && pos2.equals(player) && pos3.
equals(player)) ||
                (pos1.equals(player) && pos2.equals(EMPTY) && pos3.
equals(player))) {
            score = 100;
        } else {
            if (pos1.equals(player) && pos2.equals(EMPTY) && pos3.
equals(EMPTY) ||
                    pos1.equals(EMPTY) && pos2.equals(player) && pos3.
equals(EMPTY) ||
                    pos1.equals(EMPTY) && pos2.equals(EMPTY) && pos3.
equals(player)){
                score = 10;
            }
        }
        return score;
    }
...
    public String toKey() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                sb.append(board[i][j]);
            }
        }
        return sb.toString();
    }
}
```

Board 类提供了三个主要的函数。Board 类将棋盘封装在一个多维字符串数组中作为类成员变量。然后提供函数用来生成子节点棋盘状态（如 nextBoards()），决定个一个游戏是否结束（例如，isEndState()），最后，提供了一个方法在指定玩家的情况下，对当前棋盘

状态计算分值（例如，nextBoards（player），以及它支持的方法）。

注意 Board 类还提供了一个 toKey() 方法。这个 key 唯一地表示了一个棋盘状态，是我们的持久化存储机制中使用的唯一标识符。在这个例子中，唯一标识符是简单地将棋盘上每个格子的分值串联起来组成的值。

为了完整地表达游戏状态，我们还需要知道当前轮到了哪个玩家。因此，我们用一个高级对象封装了棋盘状态和当前的玩家，即 GameState 对象，如下面代码段所示：

```
public class GameState implements Serializable {
private Board board;
    private List<Board> history;
    private String player;

...

    public String toString(){
        StringBuilder sb = new StringBuilder('GAME [');
        sb.append(board.toKey()).append(']');
        sb.append(': player(').append(player).append(')\n');
        sb.append('   history [');
        for (Board b : history){
            sb.append(b.toKey()).append(',');
        }
        sb.append(']');
        return sb.toString();
    }
}
```

这个类中唯一让人意外的是 history 变量，这个变量记录下了游戏树中当前节点的所有祖先节点的棋盘状态。游戏树上这个记录下来的路径，在使用叶子节点的分值对祖先的分值进行更新时会用到。

最后，我们使用 Player 类来表示游戏玩家，如下面代码所示：

```
public class Player {
    public static String next(String current){
        if (current.equals('X')) return 'O';
        else return 'X';
    }
}
```

6.3.2 检视 Recursive Topology

使用前面描述的数据模型，我们可以建立一个 topology 用来向下递归游戏树。用

RecursiveTopology 类来实现。这个 topology 的代码如下所示：

```
public class RecursiveTopology {

    public static StormTopology buildTopology() {
        LOG.info('Building topology.');
        TridentTopology topology = new TridentTopology();

        // Work Queue / Spout
        LocalQueueEmitter<GameState> workSpoutEmitter =
new LocalQueueEmitter<GameState>('WorkQueue');
        LocalQueueSpout<GameState> workSpout =
new LocalQueueSpout<GameState>(workSpoutEmitter);
        GameState initialState =
new GameState(new Board(),
new ArrayList<Board>(), 'X');
        workSpoutEmitter.enqueue(initialState);

        // Scoring Queue / Spout
        LocalQueueEmitter<GameState> scoringSpoutEmitter =
new LocalQueueEmitter<GameState>('ScoringQueue');

        Stream inputStream =
topology.newStream('gamestate', workSpout);

        inputStream.each(new Fields('gamestate'),
new isEndGame())
                .each(new Fields('gamestate'),
                    new LocalQueuerFunction<GameState>(scoringSpoutEmi
tter),                    new Fields(''));

        inputStream.each(new Fields('gamestate'),
new GenerateBoards(),
new Fields('children'))
            .each(new Fields('children'),
                    new LocalQueuerFunction<GameState>(workSpoutEmitt
er),
                    new Fields());

        return topology.build();
    }
...
}
```

第一段代码配置了工作和计算分值的内存队列。读入的数据流由单独一个 spout 从 Work Queue 中读取生成。这个队列递归的种子是最初的游戏状态。

数据流然后分成两支。第一个分支的数据被过滤，只留下游戏结束的棋盘状态，这些

数据会传递到 Scoring Queue 中。第二个分支的数据用来生成新的棋盘状态，把所有后代节点都写入 Worker Queue 中。

6.3.3 队列交互

在这个示例的实现中，我们使用了内存队列。在实际生产系统中，我们会依赖 Kafka spout。下面所列的代码片段显示了 LocalQueueEmitter 类的实现。注意使用的队列是保存在一个 map 中的 BlockingQueue 实例，将这些队列和队列名称关联起来。这个类可以很方便地测试输入和输出都使用同一个队列的 topology（也就是递归 topology）：

```
public class LocalQueueEmitter<T> implements Emitter<Long>,
Serializable {
public static final int MAX_BATCH_SIZE=1000;
public static AtomicInteger successfulTransactions =
new AtomicInteger(0);
    private static Map<String, BlockingQueue<Object>> queues =
 new HashMap<String, BlockingQueue<Object>>();
private static final Logger LOG =
LoggerFactory.getLogger(LocalQueueEmitter.class);
    private String queueName;

    public LocalQueueEmitter(String queueName) {
        queues.put(queueName, new LinkedBlockingQueue<Object>());
        this.queueName = queueName;
    }

    @Override
    public void emitBatch(TransactionAttempt tx,
 Long coordinatorMeta, TridentCollector collector) {
        int size=0;
        LOG.debug('Getting batch for [' +
 tx.getTransactionId() + ']');
        while (getQueue().peek() != null &&
size <= MAX_BATCH_SIZE) {
            List<Object> values = new ArrayList<Object>();
            try {
                LOG.debug('Waiting on work from [' +
 this.queueName + ']:[' +
getQueue().size() + ']');
                values.add(getQueue().take());
                LOG.debug('Got work from [' +
this.queueName + ']:[' +
getQueue().size() + ']');
            } catch (InterruptedException ex) {
```

```
                // do something smart
            }
            collector.emit(values);
            size++;
        }
        LOG.info('Emitted [' + size + '] elements in [' +
            tx.getTransactionId() + '], [' + getQueue().size()
+ '] remain in queue.');
    }
...
    public void enqueue(T work) {
        LOG.debug('Adding work to [' + this.queueName +
']:[' + getQueue().size() + ']');
        if (getQueue().size() % 1000 == 0)
            LOG.info('[' + this.queueName + '] size = [' +
                getQueue().size() + '].');
        this.getQueue().add(work);
    }

    public BlockingQueue<Object> getQueue() {
        return LocalQueueEmitter.queues.get(this.queueName);
    }
...
}
```

这个类中的主要方法是实现了 Emitter 接口的 emitBatch 方法。当队列中有数据，并且当前处理的数据还没有达到一批数据量的上限时，这个方法继续从队列中读取数据。

而且，注意这个类提供了一个 enqueue() 方法。LocalQueueFunction 类会使用 enqueue() 方法完成递归。下面的代码段是 LocalQueueFunction 类的实现：

```
public class LocalQueuerFunction<T>  extends BaseFunction {
    private static final long serialVersionUID = 1L;
    LocalQueueEmitter<T> emitter;

    public LocalQueuerFunction(LocalQueueEmitter<T> emitter){
        this.emitter = emitter;
    }

    @SuppressWarnings('unchecked')
    @Override
    public void execute(TridentTuple tuple, TridentCollector
collector) {
        T object = (T) tuple.get(0);
        Log.debug('Queueing [' + object + ']');
        this.emitter.enqueue(object);
    }
}
```

注意，这个 function 中实际上使用了 spout 的 emitter 方法进行的实例化。这允许 function 直接将数据按队列写入到 spout 中。此外，这个结构在开发递归 topology 的时候有用，但生产环境的 topology 更倾向于使用持久化存储。没有持久化存储的情况下，因为 tuple 在处理（递归）完成前就会进行应答确认，可能会出现数据丢失。

6.3.4 function 和 filter

现在，我们将注意力集中到这个 topology 所用到的 function 和 filter 的详情上。第一个用到的简单的 filter 是用来过滤出游戏结束的棋盘状态。IsEndGame filter 的代码如下所示：

```
public class IsEndGame extends BaseFilter {
...
    @Override
    public boolean isKeep(TridentTuple tuple) {
        GameState gameState = (GameState) tuple.get(0);
        boolean keep = (gameState.getBoard().isEndState());
        if (keep){
            LOG.debug('END GAME [' + gameState + ']');
        }
        return keep;
    }
}
```

注意，如果 Trident 已经支持在一个 function 中向不同的数据流发送 tuple 的话，这个类就不是必需的了。在下面的 GenerateBoards function 中，执行了相同的检查 / 过滤功能：

```
public class GenerateBoards extends BaseFunction {

    @Override
    public void execute(TridentTuple tuple,
TridentCollector collector) {
        GameState gameState = (GameState) tuple.get(0);
        Board currentBoard = gameState.getBoard();
        List<Board> history = new ArrayList<Board>();
        history.addAll(gameState.getHistory());
        history.add(currentBoard);

        if (!currentBoard.isEndState()) {
            String nextPlayer =
                Player.next(gameState.getPlayer());
            List<Board> boards =
                gameState.getBoard().nextBoards(nextPlayer);
```

```
            Log.debug('Generated [' + boards.size() +
'] children boards for [' + gameState.toString() +
']');
            for (Board b : boards) {
                GameState newGameState =
new GameState(b, history, nextPlayer);
                List<Object> values = new ArrayList<Object>();
                values.add(newGameState);
                collector.emit(values);
            }
        } else {
            Log.debug('End game found! [' + currentBoard + ']');
        }
    }
}
```

这个 function 将当前棋盘状态添加到历史列表中，然后将一个带有子节点棋盘状态的新的 GameState 对象加入队列。

提示 也可以将 IsEndGame 实现成一个 function，添加另外一个字段来捕获结果；然而，使用上面这样实现作为例子有助于激励开发者实现 function 发送多数据流的功能。

下面代码是 Recursive Topology 的示例输出：

```
2013-12-30 21:53:40,940-0500 | INFO [Thread-28] IsEndGame.isKeep(20) |
END GAME [GAME [XXO X OOO]: player(O)
   history [          ,       O  ,    X O  ,    X OO ,X   X OO ,X O X OO
,XXO X OO ,]]
2013-12-30 21:53:40,940-0500 | INFO [Thread-28] IsEndGame.isKeep(20) |
END GAME [GAME [X OXX OOO]: player(O)
   history [          ,       O  ,    X O  ,    X OO ,X   X OO ,X O X OO
,X OXX OO ,]]
2013-12-30 21:53:40,940-0500 | INFO [Thread-28] LocalQueueEmitter.
enqueue(61) | [ScoringQueue] size = [42000]
```

6.3.5 研究 Scoring Topology

Scoring Topology 就比较简单了，因为它是线性的。稍微复杂的部分是状态更新时需要避免写之前读取的竞争情况。

Topology 代码如下：

```
public static StormTopology buildTopology() {
TridentTopology topology = new TridentTopology();

GameState exampleRecursiveState =
 GameState.playAtRandom(new Board(), 'X');
LOG.info('SIMULATED STATE : [' + exampleRecursiveState + ']');

// Scoring Queue / Spout
LocalQueueEmitter<GameState> scoringSpoutEmitter =
new LocalQueueEmitter<GameState>('ScoringQueue');
scoringSpoutEmitter.enqueue(exampleRecursiveState);
LocalQueueSpout<GameState> scoringSpout =
new LocalQueueSpout<GameState>(scoringSpoutEmitter);

Stream inputStream =
topology.newStream('gamestate', scoringSpout);

inputStream.each(new Fields('gamestate'), new IsEndGame())
                .each(new Fields('gamestate'),
                        new ScoreFunction(),
                        new Fields('board', 'score', 'player'))
                .each(new Fields('board', 'score', 'player'),
new ScoreUpdater(), new Fields());
return topology.build();
}
```

有两个 function：ScoreFunction 和 ScoreUpdater。ScoreFunction 对当前棋盘状态计算分值然后将分值发送给它所有历史记录中的棋盘状态。

ScoreFunction 的代码片段如下：

```
public class ScoreFunction extends BaseFunction {

@Override
public void execute(TridentTuple tuple,
TridentCollector collector) {
        GameState gameState = (GameState) tuple.get(0);
        String player = gameState.getPlayer();
        int score = gameState.score();

        List<Object> values = new ArrayList<Object>();
        values.add(gameState.getBoard());
        values.add(score);
        values.add(player);
        collector.emit(values);

        for (Board b : gameState.getHistory()) {
            player = Player.next(player);
```

```
            values = new ArrayList<Object>();
            values.add(b);
            values.add(score);
            values.add(player);
            collector.emit(values);
        }
    }
}
```

这个 fucntion 简单地对当前棋盘状态求分，然后为当前棋盘状态发送一个 tuple。然后，function 循环对每个历史棋盘状态进行操作，每个棋盘状态发送一个带着玩家信息 tuple，每个回合切换一次玩家。

最后来介绍 ScoreUpdater function。有一次，因为示例的原因简化了代码。下面是这个类的代码：

```
public class ScoreUpdater extends BaseFunction {
...
private static final Map<String, Integer> scores =
 new HashMap<String, Integer>();
private static final String MUTEX = 'MUTEX';

@Override
public void execute(TridentTuple tuple,
TridentCollector collector) {
    Board board = (Board) tuple.get(0);
    int score = tuple.getInteger(1);
    String player = tuple.getString(2);
    String key = board.toKey();
    LOG.debug('Got (' + board.toKey() + ') => [' + score +
'] for [' + player + ']');
    // Always compute things from X's perspective
    // We'll flip things when we interpret it if it is O's turn.
    synchronized(MUTEX){
        Integer currentScore = scores.get(key);
        if (currentScore == null ||
(player.equals('X') && score > currentScore)){
            updateScore(board, score);
        } else if (player.equals('O') &&
score > currentScore){
            updateScore(board, score);
        }
    }
}

    public void updateScore(Board board, Integer score){
```

```
            scores.put(board.toKey(), score);
            LOG.debug('Updating [' + board.toString() +
'']=>[' + score + ']');
        }
    }
```

解决写入前读取

注意上面的代码中，我们使用了互斥变量 mutex 来实现分值更新的序列性，从而消除了前面提到的竞争情况。这么做只在单机 / 本地 JVM 环境下有效。当 topology 分发到实际集群时，这个机制会失效；然而，我们只有很少的可选项来解决这个问题。

分布式锁

在其他章节中看到的，可以利用分布式锁机制比如 ZooKeeper 来实现。这种实现中，ZooKeeper 提供了一个多机器维护互斥变量 mutex 的方法。这当然是一种可行的方式，但是分布式锁会带来性能的损耗。实际上发生冲突的几率很小，这种实现中每次操作都需要进行协调操作。

陈旧时重试

另外一种有用的模式是陈旧时重试的方法。在这种场景中，对数据都会带上一个版本号，时间戳或校验码。当执行一个冲突的更新操作时，当元数据发生了改变时（例如，在 SQL/CQL 范例的 UPDATE 语句中添加 WHERE 条件），数据所包含的版本号 / 时间戳 / 校验码冲突都会使这次更新操作失败。如果元数据已经改变，它会指出我们决策基于的值已经过期了，需要重新选择数据。

显然，这些实现方法中都有权衡利弊。采用重试的方法，在极端情况下可能有大量冲突同时出现，一个线程需要重试多次才能提交更新。然而，在分布式环境中，如果一个线程阻塞住，会导致超时问题，失去了到该服务器的连接，甚至彻底故障。

近期，这个领域已经有了新的进展。建议读者了解 Paxos 和 Cassandra 使用的解决冲突更新的算法。参见下列 URL：

- http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
- http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0

在我们简单引用场景中，我们非常幸运，实际上可以把逻辑与更新操作合并在一起：

```
UPDATE gametree SET score=7 WHERE
boardkey = '000XX OXX' AND score <=7;
```

因为已经解决了写入前读取的问题，这个topology已经适合于计算Recursive Topology生成的队列中所有棋盘状态。这个topology给游戏结束的棋盘状态赋一个值，然后将该值沿着游戏树向上更新，将游戏状态对应的正确分值持久化存储。在实际的系统中，我们通过DRPC topology来访问状态数据，这样就可以预测未来多轮的结果。

执行 topology

下面是Scoring Topology的示例结果：

```
2013-12-31 13:19:14,535-0500 | INFO [main] ScoringTopology.
buildTopology(29) | SIMULATED LEAF NODE : [
---------
|X||O||X|
---------
|O||O||X|
---------
|X||X||O|
---------
] w/ state [GAME [XOXOOXXXO]: player(O)
   history [          , X       , OX       , OX  X    , OX  X  O, OX  XX O,
OXO XX O, OXO XXXO, OXOOXXXO,]]
2013-12-31 13:19:14,536-0500 | INFO [main] LocalQueueEmitter.enqueue(61)
| [ScoringQueue] size = [0].
2013-12-31 13:19:14,806-0500 | INFO [main] ScoringTopology.main(52) |
Topology submitted.
2013-12-31 13:19:25,566-0500 | INFO [Thread-24] DefaultCoordinator.
initializeTransaction(25) | Initializing Transaction [1]
2013-12-31 13:19:25,570-0500 | DEBUG [Thread-30] LocalQueueEmitter.
emitBatch(37) | Getting batch for [1]
2013-12-31 13:19:25,570-0500 | DEBUG [Thread-30] LocalQueueEmitter.
emitBatch(41) | Waiting on work from [ScoringQueue]:[1]
2013-12-31 13:19:25,570-0500 | DEBUG [Thread-30] LocalQueueEmitter.
emitBatch(43) | Got work from [ScoringQueue]:[0]
2013-12-31 13:19:25,571-0500 | DEBUG [Thread-30] LocalQueueEmitter.
emitBatch(41) | Waiting on work from [ScoringQueue]:[0]
2013-12-31 13:19:25,571-0500 | INFO [Thread-28] IsEndGame.isKeep(20) |
END GAME [GAME [XOXOOXXXO]: player(O)
   history [          , X       , OX       , OX  X    , OX  X  O, OX  XX O,
OXO XX O, OXO XXXO, OXOOXXXO,]]
```

```
...
 ScoreUpdater.updateScore(43) | Updating [
---------
| ||O||X|
---------
|O|| ||X|
---------
|X||X||O|
---------
]=>[0]
2013-12-31 13:19:25,574-0500 | DEBUG [Thread-28] ScoreUpdater.execute(27)
| Got ( OXOOXXXO) => [0] for [X]
2013-12-31 13:19:25,574-0500 | DEBUG [Thread-28] ScoreUpdater.
updateScore(43) | Updating [
---------
| ||O||X|
---------
|O||O||X|
---------
|X||X||O|
---------
]=>[0]
```

最后求解到了一个平局的叶节点，在上面输出的开始的部分。会看到这个分值通过游戏树向上传播到所有的祖先节点，根据当前的分值对这些节点进行了更新。

穷举游戏树

将 Recursive Topology 和 Scoring Topology 结合，它们会一起工作，尽可能地穷举出整个游戏空间。最可能的是，这个处理过可能会结合启发算法只存储关键节点。我们可以使用启发算法剪裁搜索空间，以减少需要估值的棋盘状态。然而无论如何，我们需要通过接口和系统之间的交互来决定当前游戏状态下最好的走法。这是我们下一节将会处理的事情。

6.3.6 分布式远程命令调用（DRPC）

现在我们已经有了一个功能性的 Recursive Topology，它会持续的查询计算整个游戏树，现在让我们来看一下同步调用问题。由 Storm 提供的 DRPC 的功能被移植到了 Trident 中，并且已经从原生 Storm 中废弃了。这个功能是在本例中使用 Trident 的主要动机。

使用 DRPC 功能构建的 topology 更常用于异步场景。图 6-5 展示了 DRPC topology。

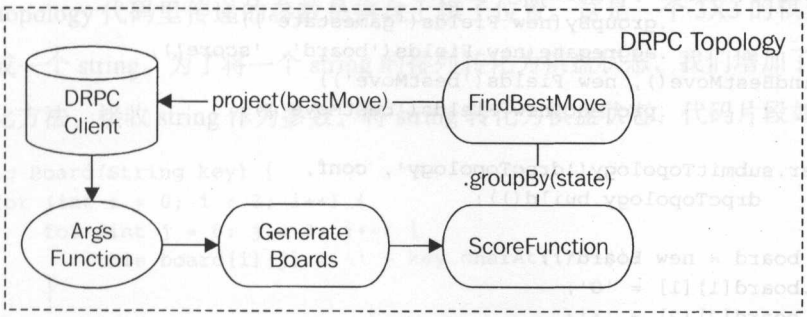


图 6-5

DRPC 客户端是一个 spout。客户端输出经过 ArgsFunction，将客户端的输入格式化，这样就可以复用当前已经存在的 funciton 了：GenerateBoard 和 ScoreFunction。然后，我们使用 .groupBy（state）并且将结果使用 Aggregater 类中的 FindBestMove 方法进行聚合。然后使用一个简单的推测，将最好的移动方式发送给客户端。

提示 你也许想了解一下 Spring Breeze，它允许你将 Plain Old Java Objects（POJOs）同时写入一个 Storm topology。这是另外一种复用方式，因为 POJO 可以通过 Web 服务进行调用，而不用引入 DRPC：https://github.com/internet-research-network/breeze

首先，看一下 topology 的代码：

```
public static void main(String[] args) throws Exception {
final LocalCluster cluster = new LocalCluster();
final Config conf = new Config();

LocalDRPC client = new LocalDRPC();
TridentTopology drpcTopology = new TridentTopology();

drpcTopology.newDRPCStream('drpc', client)
                .each(new Fields('args'),
new ArgsFunction(),
new Fields('gamestate'))
                .each(new Fields('gamestate'),
new GenerateBoards(),
new Fields('children'))
                .each(new Fields('children'),
```

```
new ScoreFunction(),
new Fields('board', 'score', 'player'))
                .groupBy(new Fields('gamestate'))
                .aggregate(new Fields('board', 'score'),
new FindBestMove(), new Fields('bestMove'))
                .project(new Fields('bestMove'));

cluster.submitTopology('drpcTopology', conf,
        drpcTopology.build());

Board board = new Board();
board.board[1][1] = 'O';
board.board[2][2] = 'X';
board.board[0][1] = 'O';
board.board[0][0] = 'X';
LOG.info('Determining best move for O on:' +
                board.toString());
LOG.info('RECEIVED RESPONSE [' +
client.execute('drpc', board.toKey()) + ']');
}
```

这个例子中，我们使用 LocalDRPC 客户端。它将一个 args 参数传递给 newDRPCStream 调用，这是 DRPC topology 的关键节点。然后，topology 像普通 topology 一样工作。

你可以看到远程调用程序实际上通过 client.execute() 方法进行。目前，这个方法接收和返回的 signature 仅支持 String。现在已经有人呼吁要求改变这个 signature。可以在这个链接看到 https://issues.apache.org/jira/browse/STORM-42。

因为该方法当前的 signature 只支持 string，我们需要将输入序列化。这个过程在 ArgsFunction 中进行，代码片段如下：

```
    @Override
    public void execute(TridentTuple tuple,
TridentCollector collector) {
        String args = tuple.getString(0);
        Log.info('Executing DRPC w/ args = [' + args + ']');
        Board board = new Board(args);
        GameState gameState =
new GameState(board, new ArrayList<Board>(), 'X');
        Log.info('Emitting [' + gameState + ']');

        List<Object> values = new ArrayList<Object>();
        values.add(gameState);
        collector.emit(values);
    }
```

我们传递给 client.execute() 的第二个参数是一个包含了我们输入的 string。在这个例子中，在 topology 代码里传递的参数是棋盘上棋子位置。这是一个 3X3 的棋盘，将每个格子组合成一个 string。为了将一个 string 的排列转化为棋盘状态，我们增加了一个 Board 类的初始化方法，接收 string 作为参数，将 string 转化为棋盘状态，代码片段如下：

```
public Board(String key) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            this.board[i][j] = '' + key.charAt(i*3+j);
        }
    }
}
```

在 DRPC topology 中，下面两个 funciton 示例了利用 DRPC 作为异步接口来获得代码复用性。这个例子中，我们单独使用这些 function。你可以想象重用更复杂的数据流也是完全可以的。

使用 GenerateBoard function，我们发射所有当前棋盘状态的子节点。然后，ScoreFunction 为每个子节点计算分值。

在 Scoring Topology 中，ScoreFunction 的输出是一个棋盘状态及其分值和玩家，这些是所有子节点棋盘状态的分值。为了决定下一步棋怎么走最好，我们只需要简单地取最大或者最小的值。这个可以使用一个简单的 Aggregator 来实现。我们建立一个名字是 FindBestMove 的聚合 function，代码如下所示：

```
public class FindBestMove extends BaseAggregator<BestMove> {
    private static final long serialVersionUID = 1L;

    @Override
    public BestMove init(Object batchId,
TridentCollector collector) {
        Log.info('Batch Id = [' + batchId + ']');
        return new BestMove();
    }

    @Override
    public void aggregate(BestMove currentBestMove,
TridentTuple tuple, TridentCollector collector) {
        Board board = (Board) tuple.get(0);
        Integer score = tuple.getInteger(1);
        if (score > currentBestMove.score){
            currentBestMove.score = score;
```

```
                currentBestMove.bestMove = board;
            }
        }

        @Override
        public void complete(BestMove bestMove,
    TridentCollector collector) {
            collector.emit(new Values(bestMove));
        }

    }
```

这个聚合类继承了 Java 范类 BaseAggregator。在这个例子中，我们想发射最好的移动步骤，以及对应的分值。因此我们将 BestMove 类作为参数传给 BaseAggregator 类，这个类很简单：

```
public class BestMove {
    public Board bestMove;
    public Integer score = Integer.MIN_VALUE;

    public String toString(){
        return bestMove.toString() + '[' + score + ']';
    }
}
```

回忆一下，进行聚合操作时，Trident 在初始化时调用 init() 方法，这个方法返回最初的聚合值。在我们的例子里，我们给 BestMove 最坏的移动步骤作为种子。注意传递给 BestMove 类的种子值是绝对最小值。然后，Trident 随后调用 aggregate() 方法，允许将 tuple 组合为聚合值。一个聚合操作还可以在这里发射数据，因为我们只需要关心最佳移动步骤，我们不用在 aggregate() 方法中发射任何数据。最后，当所有 tuple 的值都进行了聚合操作后，Trident 调用 complete() 方法。在这个方法里我们发射出最佳的移动步骤。

下面是 topology 的输出：

```
2013-12-31 13:53:42,979-0500 | INFO [main] DrpcTopology.main(43) |
Determining best move for O on:
---------
|X||O|| |
---------
| ||O|| |
---------
| || ||X|
---------
```

```
00:00  INFO: Executing DRPC w/ args = [XO  O   X]
00:00  INFO: Emitting [GAME [XO  O   X]: player(X)
    history []]
00:00  INFO: Batch Id = [storm.trident.spout.RichSpoutBatchId@1e8466d2]
2013-12-31 13:53:44,092-0500 | INFO [main] DrpcTopology.main(44) |
RECEIVED RESPONSE [[[
---------
|X||O|| |
---------
| ||O|| |
---------
| ||O||X|
---------
[10000]]]]
```

本例中，轮到O来走，他有一个获胜的机会。你可以看到，topology正确地判断出了胜利的机会，并且返回了可以取胜的最佳走法（根据分值算出）。

远程部署

我们已经展示了如何在本地环境下调用DRPC topology。为了在远程调用topology，需要启动一个DRPC服务器。启动这个服务和启动Storm中其他服务是一样的方法，用drpc作为参数执行Storm脚本：

```
bin/storm drpc
```

Storm集群会连接到DRPC服务器上接收命令。我们需要通过配置告诉Storm集群DRPC服务器的位置。在storm.yaml文件中如下配置：

```
drpc.servers:
- 'drpchost1 '
- 'drpchost2'
```

当配置了DRPC服务的服务器启动后，topology的提交方式和其他topology一样，DRPC客户端可以被其他任何需要大规模分布式异步处理的Java应用程序来使用。为了从本地DRPC客户端切换到远程，唯一需要改变的是DRPC客户端的命令。需要使用下面的代码行来替换本地DRPC客户端：

```
DRPCClient client = new DRPCClient('drpchost1', 3772);
```

参数指定了DRPC服务的服务器和端口，需要和YAML文件中的配置一致。

总结

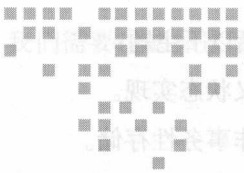
在本章中，我们实现了一个人工智能的应用场景。这个领域中有很多问题需要利用树或者图形数据结构，这些数据结构中最合适的算法往往是递归。为了示例这些算法在 Storm 中的实现，我们选择了极大极小算法，并且使用 Storm 的结构实现了它。

过程中，我们注意到 Storm 的一些固有约束导致实现的复杂度超出了预估，我们如何绕过这些约束，实现一个可用的 / 可扩展的系统所涉及的模式和实现方法。

最后，我们介绍了 DRPC 服务。DRPC 可以用来为客户端实现一个同步接口。DRPC 还允许在异步接口和同步接口之间设计可复用的代码和数据流。

将同步和异步的 topology 结合，共用状态信息，不仅仅是人工智能应用中，也是数据分析中可以使用的非常有效的模式。通常，新数据在后台持续地到来，但是用户会通过同步接口和数据进行查询。当你将 DRPC 和其他章节中提到的 Trident 状态管理功能结合起来，就可以构造一个适用于实时分析场景的系统。

下一章中，我们将非事务性实时分析系统 Druid 整合进 Storm 中。我们还将深入了解 Trident 和 ZooKeeper 的分布状态管理。



第 7 章 Chapter 7

整合 Druid 进行金融分析

本章中，我们扩展 Trident 应用来建立一个实时金融分析仪表盘。这个系统用于处理金融消息，随着时间提供各个粒度的股票价格展示。这个系统示例使用自定义状态实现来整合一个非事务系统。

在前面的例子中，我们使用 Trident 对一定时间内的事件进行累加计算。在分析单维度的数据场景下非常适用，但是它的体系结构设计得不够灵活。要引入一个新的维度就需要用 Java 开发以及发布新代码。

传统上，数据仓库技术和商业智能平台用来计算和存储维度数据进行分析。数据仓库部署作为联机分析处理系统（On-line AnalyticsProcessing，OLAP）的一部分，是从联机事务处理中分离出来的。数据通常会在一定的延迟后转发到 OLAP 系统中。这个模型可以满足对数据进行回溯分析，但满足不了对实时分析需求。

类似地，数据科学家还有其他批处理的数据处理技术。例如 PIG 这类语言可用于表达查询操作，然后查询被编译成 job，在大规模的数据集上运行。幸运的是，已经有 Hadoop 这种分布式系统的平台可以用来实现类似操作，但 Hadoop 还是会引入一定的延迟。

这些实现方法对于金融系统都有不足，因为金融系统无法接受这样的延迟。仅靠增加批处理任务速度来满足金融系统的实时分析，代价可能会很大。

本章中，我们会将 Storm 扩展为一个灵活的系统，只需要很小的代价就可以增加新的维度，同时能够提供实时分析的能力。这意味着在数据提取和有效分析之间只有很小的延时。

本章包括以下主题：

- 自定义状态实现。
- 集成非事务性存储。
- 使用 ZooKeeper 实现分布式状态。
- Druid 和实时聚合分析。

7.1 使用场景

在这个使用场景中，我们会深入金融系统中的股票订单数据。我们可以通过 REpresentational State Transfer（REST）接口获取这些信息，根据时间维度传输并且使用这些价格信息。

金融行业典型的消息格式是 Financial Information eXchange（FIX）。格式定义的详情见 http://www.fixprotocol.org/。

FIX 消息的一个例子如下：

```
23:25:1256=BANZAI6=011=135215791235714=017=520=031=032=037=538=1
000039=054=155=SPY150=2151=010=2528=FIX.4.19=10435=F34=649=BANZ
AI52=20121105-
```

FIX 消息本质上是 key-value 对的数据流。ASCII 字符 01，Start of Header（SOH），是 key-value 对的分隔符。FIX 通过标签来引用 key。比如上面的消息，标签用整数标记。每个标签包括一个字段名称和数据类型。标签类型完整参考见 http://www.fixprotocol.org/FIXimate3.0/en/FIX.4.2/fields_sorted_by_tagnum.html。

我们实例中用到的重要字段如表 7-1 所示。

表 7-1

标签 ID	字段名	描　述	数据类型
11	ClOrdID	消息的唯一标识符	String
35	MsgType	FIX 消息的类型	String
44	Price	每份股票的价格	Price
55	Symbol	股票代码	String

FIX 是一个建立在 TCP/IP 层之上的协议。在一个实际的系统中，这些消息是通过 TCP/IP 协议接收的。在 Storm 中的集成，通常应该使用 Kafka 队列来传输这些消息。但是为了简化起见，在我们的示例中，直接从一个文件中读取 FIX 消息。FIX 协议支持多种消

息类型。有些消息是控制消息（比如登录、心跳包等）。我们需要过滤掉这些无用的消息，只把包含价格信息的消息发送给分析引擎。

7.2　集成一个非事务系统

为了在我们前面的例子上进行扩展，可以开发一个框架，通过配置允许用户指定需要聚合的维度。然后，使用配置在我们的 topology 中维护一组内存数据集来积累聚合结果，但是任何内存存储都容易受到程序异常或失败的影响。为了解决容错性，之后再将聚合结果持久化存储到数据库中。

我们需要预估和支持用户可能执行的所有不同类型的聚合操作（例如，求和、均值、空间索引等），这看起来需要大量的努力。

幸运的是，实时分析引擎已经有了不少现成的选择。其中一个流行的开源项目是 Druid。下面的描述是从其白皮书中摘录出来的 http://static.druid.io/docs/druid.pdf：

Druid 是一个开源的、实时分析的数据存储机制，支持在大数据集合上执行快速临时查询。系统将一个面向列的数据布局，一个无共享的架构，以及一个高级索引结构结合起来，允许用户在亚秒时间内查询数十亿的数据表。Druid 可以水平扩展，是 Metamarkets 数据分析平台的核心引擎。

从上面的摘录来看，Druid 非常适合我们的需求。现在的挑战是如何将它和 Storm 结合起来。

Druid 的技术栈天生就适合基于 Storm 的生态系统。和 Storm 类似，它使用 ZooKeeper 来协调节点。Druid 还支持直接和 Kafka 集成。在一些场景下，这是非常方便的。在我们的例子里，会演示如何集成一个非事务系统，我们会将 Druid 和 Storm 直接整合起来。

我们这里先简要地介绍一下 Druid。关于 Druid 更详细的信息，参见下面的连接：https://github.com/metamx/druid/wiki

Druid 通过它的 Real-time 节点收集信息。基于配置的粒度不同，Real-time 节点收集事件信息到片段中，并将片段持久化到一个深度存储机制中。Druid 将这些数据片段的元数据存储在 MySQL 中。Master 节点识别新片段，基于规则找到该片段对应的 Compute 节点，然后通知该 Compute 节点拉取新片段。一个 Broker 节点位于 Compute 节点之前，接收消费者发送的 REST 查询语句，并且将查询语句分发给合适的 Compute 节点。

因此，集成了 Druid 的 Storm 结构看起来如图 7-1 所示。

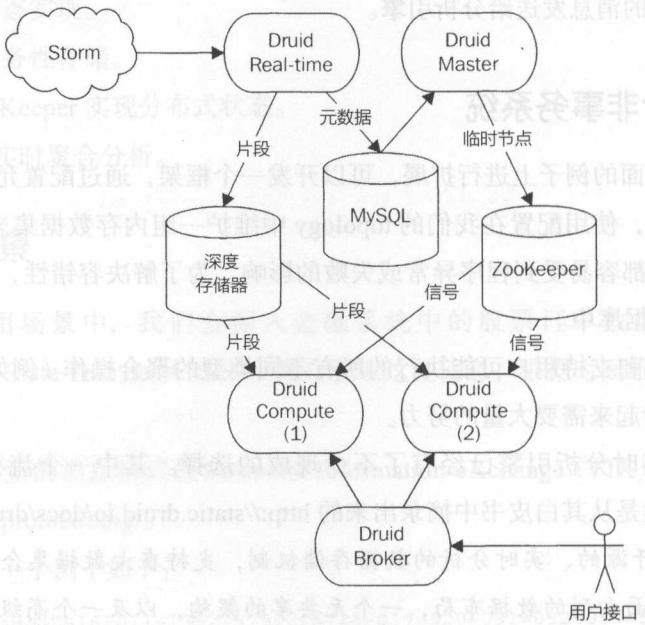


图 7-1

如图 7-1 所示，其中涉及了三种存储机制。MySQL 数据库是一个简单的元数据仓库。包括了所有数据段的元数据信息。深度存储器保存了实际的数据片段信息。每个片段包括了一个特定时段内包含事件的合并索引，索引是基于配置文件中定义的维度和聚合生成的。这样，每个片段的文件可以非常大（比如，2G 的块）。在我们的例子中，我们使用 Cassandra 作为深度存储器。

第三种存储机制是 ZooKeeper。ZooKeeper 中的存储是事务性的，只是用来存储控制信息。当一个新的数据片段可用时，Master 节点在 ZooKeeper 写入一个临时的节点。Conpute 节点订阅同样的路径，临时节点触发该 Conpute 节点拉取新的数据片段。在数据片段成功接收完毕后，Compute 节点从 ZooKeeper 中移除临时节点。

我们例子中，整个事件的顺序如图 7-2 所示。

图 7-2 展示出了 Storm 下游事件处理的流程。重要的是认识到实时分析引擎没有能力对事务处理进行回滚操作。分析系统为了高度优化处理速度和聚合操作，对应的牺牲了事务的完整性。

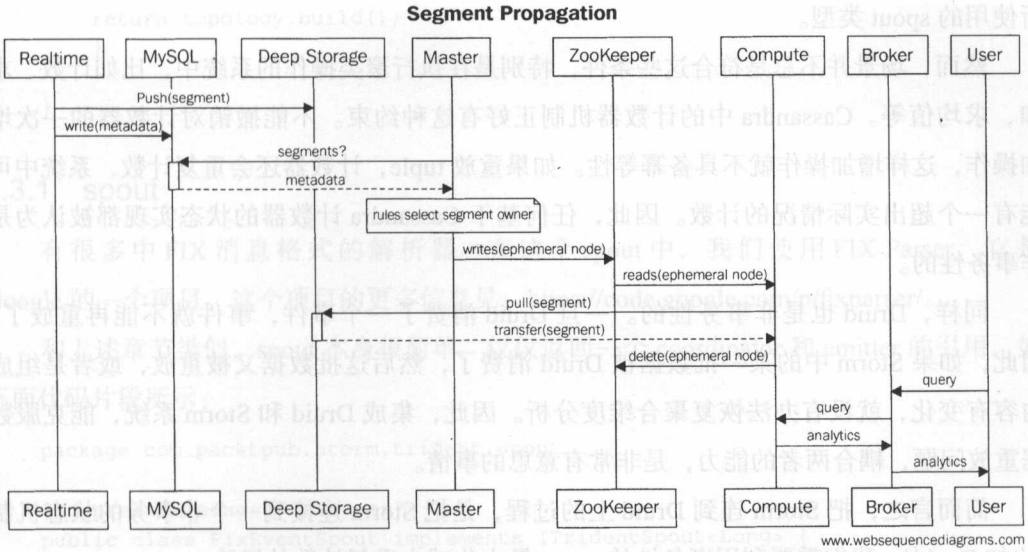


图　7-2

回忆 Trident 的状态分类，存在三种不同类型的状态：事务的、不透明的、非事务的。一个事务型的状态要求每批数据的内容要一直保存到事务结束。一个不透明事务状态可以容忍一批数据组成内容的变化。最后是非事务型状态，不保证上述任意一种语义。

总结 storm.trident.state.State 对象的 Javadoc，有三种不同的状态，参见表 7-2。

表　7-2

状　态	说　明
非事务型状态	这个状态中，commit 操作被忽略。不能进行事务回滚操作。更新操作是永久的
重复事务型状态	系统在一批数据处理确认之前是幂等的
不透明事务型状态	状态转换是递增的。前面的存储的状态会附带有批次编号，能够容忍数据重发是批次数据组成的变化

要意识到很重要的一点，在 topology 中引入状态可以很方便地将任何数据序列化写入到存储中。这可能对性能产生较大影响。如果可能，最好将整个系统实现为幂等的。如果所有写操作是幂等的，你根本不需要引入事务存储（或者状态），因为幂等体系结构自然能够对数据重放进行容错。

通常，如果你可以控制状态存储后端的数据库中数据格式，你可以调整格式添加额外的信息参与到事务处理中：为了重放事务存储上一次提交数据批次的标识符，以及不透明事务中前一个状态信息。在状态的实现中，就可以利用这些信息保证状态对象能适应前端

所使用的 spout 类型。

然而，场景并不总是符合这些条件，特别是在执行聚类操作的系统中，比如计数、求和、求均值等。Cassandra 中的计数器机制正好有这种约束。不能撤销对计数器的一次增加操作，这样增加操作就不具备幂等性。如果重放 tuple，计数器还会重复计数，系统中可能有一个超出实际情况的计数。因此，任何基于 Cassandra 计数器的状态实现都被认为是非事务性的。

同样，Druid 也是非事务性的。一旦 Druid 消费了一个事件，事件就不能再重放了。因此，如果 Storm 中的某一批数据被 Druid 消费了，然后这批数据又被重放，或者是组成内容有变化，就没有办法恢复聚合维度分析。因此，集成 Druid 和 Storm 系统，能克服数据重放问题，耦合两者的能力，是非常有意思的事情。

简而言之，把 Storm 连到 Druid 上的过程，是把 Storm 连接到一个非事务的状态机制（比如 Druid），我们需要利用事务性的 spout 最小化减少重复计数的风险。

7.3 topology

介绍完体系结构的概念，让我们回到应用场景中来。为了能将注意力放在系统集成过程中，我们将保持 topology 的简洁如图 7-3 所示。

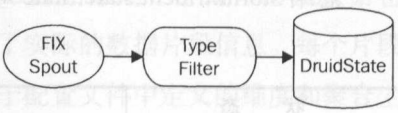


图 7-3

FIX Spout 发送包含了 FIX 消息的 tuple。然后 Filter 检查消息类型，过滤出包含了价格信息的股票定单。然后将过滤后的 tuple 发送到 DruidState 状态中，它是 Storm 系统和 Druid 系统之间的桥接。

这个简单 topology 的代码如下：

```
public class FinancialAnalyticsTopology {

    public static StormTopology buildTopology() {
    TridentTopology topology = new TridentTopology();
    FixEventSpout spout = new FixEventSpout();
    Stream inputStream =
topology.newStream("message", spout);
    inputStream.each(new Fields("message"),
new MessageTypeFilter())
        .partitionPersist(new DruidStateFactory(),
new Fields("message"), new DruidStateUpdater());
```

```
        return topology.build();
        }
    }
```

7.3.1 spout

有很多中 FIX 消息格式的解析器。在这个 spout 中，我们使用 FIX Parser，它是 Google 的一个项目。这个项目的更多信息见：https://code.google.com/p/fixparser/。

和上述章节类似，spout 本身很简单。仅仅返回一个 coordinator 和 emitter 的引用，如下面代码片段所示：

```
package com.packtpub.storm.trident.spout;

@SuppressWarnings("rawtypes")
public class FixEventSpout implements ITridentSpout<Long> {
    private static final long serialVersionUID = 1L;
    SpoutOutputCollector collector;
    BatchCoordinator<Long> coordinator = new DefaultCoordinator();
    Emitter<Long> emitter = new FixEventEmitter();
    ...
    @Override
    public Fields getOutputFields() {
        return new Fields("message");
    }
}
```

如上面代码所示，spout 声明了一个单字段的输出：message。这个消息会包含由 Emitter 生成的 FixMessageDto 对象，代码如下所示：

```
package com.packtpub.storm.trident.spout;

public class FixEventEmitter implements Emitter<Long>,
Serializable {
    private static final long serialVersionUID = 1L;
    public static AtomicInteger successfulTransactions =
new AtomicInteger(0);
    public static AtomicInteger uids = new AtomicInteger(0);

    @SuppressWarnings("rawtypes")
    @Override
    public void emitBatch(TransactionAttempt tx,
    Long coordinatorMeta, TridentCollector collector) {
    InputStream inputStream = null;
    File file = new File("fix_data.txt");
    try {
```

```
            inputStream =
    new BufferedInputStream(new FileInputStream(file));
            SimpleFixParser parser = new SimpleFixParser(inputStream);
            SimpleFixMessage msg = null;
            do {
            msg = parser.readFixMessage();
            if (null != msg) {
                FixMessageDto dto = new FixMessageDto();
                for (TagValue tagValue : msg.fields()) {
                    if (tagValue.tag().equals("6")) { // AvgPx
                        // dto.price =
    //Double.valueOf((String) tagValue.value());
                        dto.price =
                        new Double((int) (Math.random() * 100));
                    } else if (tagValue.tag().equals("35")) {
                        dto.msgType = (String)tagValue.value();
                    } else if (tagValue.tag().equals("55")) {
                       dto.symbol = (String) tagValue.value();
                    } else if (tagValue.tag().equals("11")){
                       // dto.uid = (String) tagValue.value();
                       dto.uid =
                       Integer.toString(uids.incrementAndGet());
                    }
                }
                new ObjectOutputStream(
                new ByteArrayOutputStream()).writeObject(dto);
                List<Object> message = new ArrayList<Object>();
                message.add(dto);
                collector.emit(message);
            }
        } while (msg != null);
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            IoUtils.closeSilently(inputStream);
        }
        }

        @Override
        public void success(TransactionAttempt tx) {
            successfulTransactions.incrementAndGet();
        }

        @Override
        public void close() {
        }
    }
```

从前面的代码中可以看到，我们每批数据都会重复解析一个文件。开始说过，一个实时系统可能通过 TCP/IP 传输数据并且将数据写入 Kafka 队列。然后我们使用 Kafka spout

发射消息。这个方案常常是一种优先选择；但是为了彻底封装 Storm 中的数据处理，系统大部分情况下会尽可能将原始消息文本入队列。在上面的设计中，我们在一个 function 中解析文本，而不是在一个 spout 中解析。

虽然这个 spout 仅适用于本例，要注意每批数据的组成都是相同的。特别是每批数据都包括了一个文件中的所有消息。因为我们状态实现的设计依赖于这个特性，在实际系统中，就需要使用 TransactionalKafkaSpout。

7.3.2　filter

和 spout 类似，filter 也很简单。它检查 msgType 对象，并且过滤掉不包含订单的消息。含有订单的消息是有效的股票购买额度，它们包括了一个交易股票的平均价格购买股票的文号。下面是过滤器的代码：

```
package com.packtpub.storm.trident.operator;

public class MessageTypeFilter extends BaseFilter {
    private static final long serialVersionUID = 1L;

    @Override
    public boolean isKeep(TridentTuple tuple) {
        FixMessageDto message = (FixMessageDto) tuple.getValue(0);
    if (message.msgType.equals("8")) {
        return true;
    }
    return false;
    }
}
```

这提供了一个好机会可以看到在 Storm 中数据序列化的重要性。注意上面代码中，过滤器在 FixMessageDto 对象上操作。如果能使用 SimpleFixMessage 对象的话会更简单，但是 SimpleFixMessage 对象是不可序列化的。这在本机环境中不会引起问题。但是，集群模式中 tuple 数据会在不同的机器之间传输，tuple 中所有的元素都必须是可序列化的。

开发人员常常向 tuple 内提交非序列化的数据对象。这会引起后续发布时的错误。为了保证 tuple 中的对象都是可序列化的，添加单元测试来验证这些对象是否可序列化。测试非常简单，用下面的代码：

```
new ObjectOutputStream(
new ByteArrayOutputStream()).
writeObject(YOUR_OBJECT);
```

7.3.3 状态设计

现在，让我们继续探索这个例子中最有趣的部分。为了集成 Druid 和 Storm，需要嵌入一个实时分析的 Druid 服务到我们的 topology 中，并且实现必须的接口将数据流连接给它。为了缓解连接到一个非事务系统继承来的风险，我们利用了 ZooKeeper 来维护状态信息。这个持久化存储不能避免程序失败导致的异常，但它会帮助确认哪些数据受到了程序失败的影响。

上层的设计如图 7-4 所示。

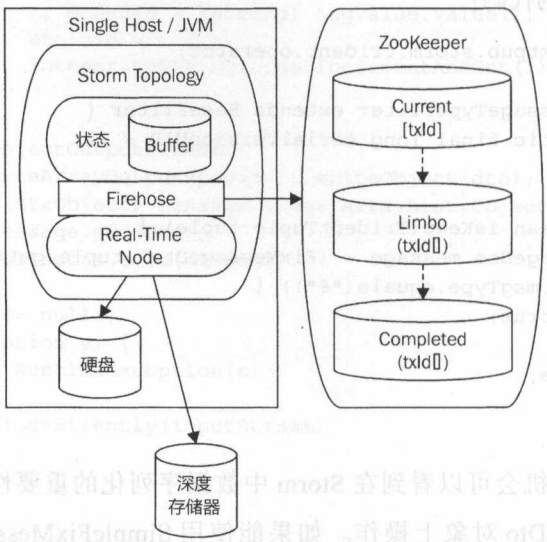


图 7-4

在上层，Storm 使用工厂类在 worker 的 JVM 进程中建立了一个状态对象。每批数据都会建立一个状态对象。状态工厂对象保证在它返回任何状态对象之前，实时服务都在运行，如果实时服务没有运行则启动该服务。然后状态对象被缓存起来直到 Storm 调用 commit 方法。当 Storm 调用 commit 时，状态对象解锁 Druid Firehose。这样会发送信号到 Druid 说明数据已经准备好做聚合操作了。然后我们将 Storm 阻塞在 commit 方法中，同时实时服务开始通过 Firehose 拉取数据。

为了保证每个数据分片最少处理一次，我们给每个数据分片关联一个分片标识符。分片标识符是由数据批次标识符和分片索引组成的，因为我们使用了一个事务型 spout，分片标识符能够唯一地标识一个数据集。

Firehose 将标识符持久化到 ZooKeeper 中来维护分片的状态。

在 ZooKeeper 中有三个状态，参见表 7-3。

表　7-3

状　态	描　述
inProgress	这个 ZooKeeper 路径保存了 Druid 正在处理的分片标识符
Limbo	这个 ZooKeeper 路径保存了 Druid 已经消费完成了的分片的标识符，但这些分片还没有提交
Completed	这个 ZooKeeper 路径保存了 Druid 已经成功提交了的分片的标识符

当一批数据在处理中时，Firehose 将分片标识符写入到 inProgress 路径下。当 Druid 已经将一个 Storm 的分片数据都读取出来以后，分片标识符会移动到 Limbo 中，并且仅当接收到 Druid 发出的 commit 消息时，就会释放 Storm 的阻塞状态，以继续数据处理。

当接收到 Druid 发送来的 commit 消息时，Firehose 将分片节点移动到 Completed 路径下。这个时候，我们假设数据已经写到硬盘上了。但还是有可能在硬盘写入失败的情况下丢失数据。然而，如果我们假设可以使用批处理系统重现聚类操作，这种硬盘失败就是可接受的风险。

图 7-5 中的状态机捕获了处理中的不同阶段。

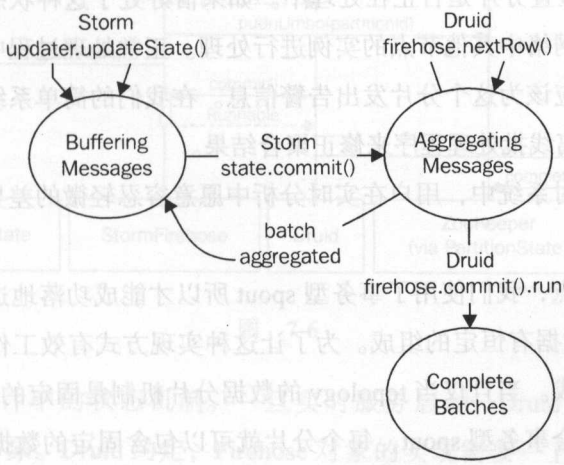


图　7-5

如图 7-5 所示，在 Buffering Messages 和 Aggregating Messages 之间有一个循环。主控循环在这两个状态中快速切换，将时间划分给 Storm 处理循环和 Druid 聚类循环。状态是相互排斥的：系统要么正在对一批数据做聚类操作，要么正在缓存下一批数据。

当 Druid 将信息写入硬盘时触发第三个状态。触发第三个状态的时候（后面会讲到），Firehose 会收到通知，我们可以更新持久化存储机制来表明这批数据已经安全处理完成。这批被 Druid 消费的数据在调用 commit 操作之前，必须一直留在 Limbo 状态。

在 Limbo 中，无法得知数据的处理状态。Druid 可能已经聚类出结果，也可能没有。

在遇到处理失败的事件时，Storm 可以利用其他 TridentState 实例来完成处理过程。因此，对每个数据分片，Firehose 必须执行下面几个步骤：

1. Firehose 必须检查分片是否已经处理完成。如果已经完成的话，这个分片就是重放的数据，可能是因为下游的故障。因为每批数据能够保证和以前一样，因此，在 Druid 聚类完成它的内容之后，Druid 就可以安全地将这批数据忽略。系统可能会输出一条告警信息。

2. Firehose 必须检查分片是否在 Limbo 状态中。如果分片标识在 Limbo 状态中，Druid 已经消费了分片，但还没有提交或者在提交后在 Firehose 更新 ZooKeeper 之前系统出现了错误。系统应该发出一个告警信息。不应该尝试再处理这批数据，因为它已经被 Druid 完全消费并且我们不知道聚合操作的状态。它简单地返回，让 Storm 继续处理下一批数据。

3. Firehose 必须检查分片是否正在处理中。如果恰好处于这种状态，那么分片数据因为某种原因，正在被网络中其他节点的实例进行处理。正常处理过程中不应该发生这种情况。在本例中，系统应该为这个分片发出告警信息。在我们的简单系统中，我们简单的处理，将这种数据流给离线批处理程序来修正聚合结果。

在很多大规模实时系统中，用户在实时分析中愿意容忍轻微的差异，只要偏差很少出现并且能够快速恢复。

注意很重要的一点，我们使用了事务型 spout 所以才能成功落地这个实现方式。事务型 spout 保证了每批数据有恒定的组成。为了让这种实现方式有效工作，每批数据的每个分片必须有恒定的组成。当且仅当 topology 的数据分片机制是固定的才能满足这种要求。有固定的分片机制配合事务型 spout，每个分片就可以包含固定的数据，甚至在数据重放的时候也是如此。我们之前选用的随机分组（shuffle group）就无法满足这种场景。本例中

的 topology 具有确定性。这保证了每批数据包含了一个批次标识符，再结合分片索引，不同时间都可以表示了恒定的数据集合。

7.4　实现体系结构

设计完成后，我们现在开始关注实现细节。图 7-6 展示了实现的时序图。

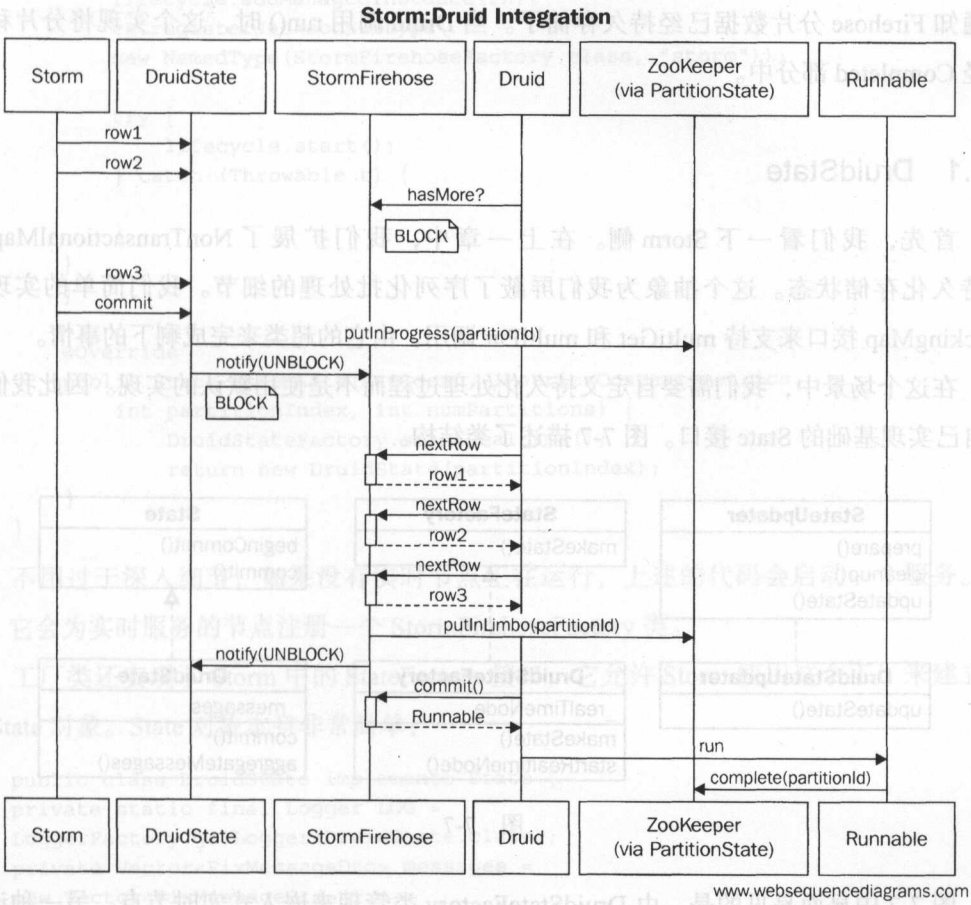


图　7-6

图 7-6 实现了设计中的状态机制。一旦实时服务启动，Druid 使用 hasMore() 方法轮询 StormFirehose 对象。Druid 约定，Firehose 对象的实现需要一直阻塞直到有数据到来。当 Druid 轮询并且 Firehose 对象被阻塞时，Storm 发送 tuple 到 DruidState 对象的消

息缓存中。当一批数据发送完成后，Storm 调用 DruidState 上的 commit() 方法。这时候，PartitionStatus 分片状态更新。分片开始处理，Druid 的实现通知取消 StormFirehose 对象的阻塞状态。

Druid 开始从 StormFirehose 对象通过 nextRow() 方法拉取数据。当 StormFirehose 对象消费完分片数据的内容时，它将分片放入 limbo 路径，并且释放控制给 Storm。

最后，当调用了 StormFirehose 中的 commit 方法时，返回一个 Runnable，Druid 用它来通知 Firehose 分片数据已经持久存储了。当 Druid 调用 run() 时，这个实现将分片移到已经 Completed 部分中。

7.4.1 DruidState

首先，我们看一下 Storm 侧。在上一章中，我们扩展了 NonTransactionalMap 类来持久化存储状态。这个抽象为我们屏蔽了序列化批处理的细节。我们简单的实现了 IBackingMap 接口来支持 multiGet 和 multiPut 调用，由它的超类来完成剩下的事情。

在这个场景中，我们需要自定义持久化处理过程而不是使用默认的实现。因此我们需要自己实现基础的 State 接口。图 7-7 描述了类结构。

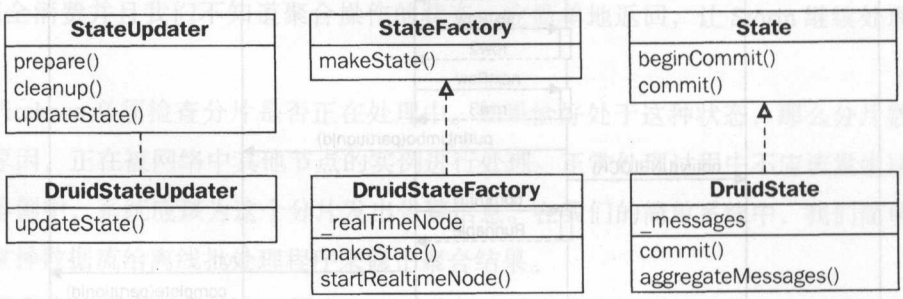


图 7-7

图 7-7 中显而易见的是，由 DruidStateFactory 类管理来嵌入式实时节点。另一种论点是可以在 updater 中管理嵌入式实时服务。然而，因为每个 JVM 中只会有一个实时服务的实例，并且实例需要在所有状态对象前生成，因此看起来把嵌入式服务的生命周期管理放在工厂类中更自然一些。

下面代码片段包括了 DruidStateFactory 类中的相关部分：

```
public class DruidStateFactory implements StateFactory {
    private static final long serialVersionUID = 1L;
    private static final Logger LOG =
LoggerFactory.getLogger(DruidStateFactory.class);
    private static RealtimeNode rn = null;

    private static synchronized void startRealtime() {
    if (rn == null) {
        final Lifecycle lifecycle = new Lifecycle();
        rn = RealtimeNode.builder().build();
        lifecycle.addManagedInstance(rn);
        rn.registerJacksonSubtype(
        new NamedType(StormFirehoseFactory.class, "storm"));

        try {
            lifecycle.start();
        } catch (Throwable t) {

        }
    }
    }

    @Override
    public State makeState(Map conf, IMetricsContext metrics,
        int partitionIndex, int numPartitions) {
            DruidStateFactory.startRealtime();
            return new DruidState(partitionIndex);
    }
}
```

不用过于深入细节，如果没有实时节点正在运行，上述的代码会启动一个服务。并且，它会为实时服务的节点注册一个 StormFirehoseFactory 类。

工厂类还实现了 Storm 中的 StateFactory 接口，它允许 Storm 使用这个工厂来建立新的 State 对象。State 对象本身非常简单：

```
public class DruidState implements State {
private static final Logger LOG =
LoggerFactory.getLogger(DruidState.class);
private Vector<FixMessageDto> messages =
new Vector<FixMessageDto>();
    private int partitionIndex;

public DruidState(int partitionIndex){
    this.partitionIndex = partitionIndex;
}
```

```
@Override
    public void beginCommit(Long batchId) {
}

@Override
public void commit(Long batchId) {
    String partitionId = batchId.toString() + "-" +
                         partitionIndex;
    LOG.info("Committing partition [" +
        partitionIndex + "] of batch [" + batchId + "]");
    try {
        if (StormFirehose.STATUS.isCompleted(partitionId)) {
        LOG.warn("Encountered completed partition ["
            + partitionIndex + "] of batch [" + batchId
                + "]");
        return;
    } else if (StormFirehose.STATUS.isInLimbo(partitionId)) {
        LOG.warn("Encountered limbo partition [" + partitionIndex
                + "] of batch [" + batchId +
                "] : NOTIFY THE AUTHORITIES!");
        return;
    } else if (StormFirehose.STATUS.isInProgress(partitionId)) {
            LOG.warn("Encountered in-progress partition [\" +
            partitionIndex + \"] of batch [" + batchId +
            "] : NOTIFY THE AUTHORITIES!");
        return;
    }
    StormFirehose.STATUS.putInProgress(partitionId);
    StormFirehoseFactory.getFirehose()
        .sendMessages(partitionId, messages);
    } catch (Exception e) {
            LOG.error("Could not start firehose for [" +
                        partitionIndex + "] of batch [" +
                        batchId + "]", e);
    }
    }

public void aggregateMessage(FixMessageDto message) {
    messages.add(message);
}
}
```

上面的代码中可以看到，State 对象是一个消息缓存。我们下面立即会了解到，它将实际的 commit 逻辑委托给 Firehose 对象。然而，这个类中有几行关键代码实现了我们前面提到的错误发现机制。

State 对象 Commit() 方法中的条件逻辑检查 ZooKeeper 的状态来决定当前分片数

据是否已经成功地处理过（inCompleted），提交失败（inLimbo），或者在处理中失败了（inProgress）。我们在检视 DruidPartitionStatus 对象时会深入了解状态存储实现。

还有一点需要注意的是，commit() 方法是由 Storm 直接调用的，但是 aggregateMessage() 方法是由 updater 来调用的。即使 Storm 从来不会并发调用这些方法，我们仍然使用了线程安全的容器类型。

DruidStateUpdater 的代码如下所示：

```
public class DruidStateUpdater implements StateUpdater<DruidState> {
...
@Override
public void updateState(DruidState state,
List<TridentTuple> tuples, TridentCollector collector) {
for (TridentTuple tuple : tuples) {
      FixMessageDto message = (FixMessageDto) tuple.getValue(0);
      state.aggregateMessage(message);
    }
}
}
```

如前面代码所示，updater 简单的循环处理 tuple，并将它们传递到状态对象的缓存中。

7.4.2　实现 StormFirehose 对象

在我们开始关注 Druid 侧的实现之前，我们应该退一步先来更详细地讨论一下 Druid。Druid 消息通过特殊的文件进行配置。我们的例子中，这个文件是 realtime.spec，如下面代码所示：

```
[{
    "schema": {
        "dataSource": "stockinfo",
        "aggregators": [
            { "type": "count", "name": "orders"},
            { "type": "doubleSum", "fieldName": "price", "name":
            "totalPrice" }
        ],
        "indexGranularity": "minute",
        "shardSpec": {"type": "none"}
    },

    "config": {
        "maxRowsInMemory": 50000,
        "intermediatePersistPeriod": "PT30s"
    },
```

```
    "firehose": {
        "type": "storm",
        "sleepUsec": 100000,
        "maxGeneratedRows": 5000000,
        "seed": 0,
        "nTokens": 255,
        "nPerSleep": 3
    },

    "plumber": {
        "type": "realtime",
        "windowPeriod": "PT30s",
        "segmentGranularity": "minute",
        "basePersistDirectory":
        "/tmp/example/rand_realtime/basePersist"
    }
}]
```

前面配置文件中对我们示例最重要的元素是 scheme 和 firehose。schema 元素定义了数据和 Druid 要对数据执行的聚合操作。例子中，Druid 会对 order 字段中股票标志出现的次数进行计数，并且将股票总价记录在 totalPrice 字段中。totalPrice 字段会用来计算一段时期内股价均值。另外，需要指定一个 indexGranularity 对象来说明索引的临时粒度。

Firehose 元素包括了 Firehose 对象的配置项。如同我们在 StateFactory 接口中看到的，一个具体的实现会在实时服务启动时，在 Druid 注册一个 FirehoseFactory 对象。工厂类作为一个 Jackson 子类进行注册。当实时配置说明文件解析时，JSON 格式的 firehose 元素中的类型用来联系适当的 FirehoseFactory 去获取数据流。

关于 JSON 多态性的更多信息，参考下面网站：

http://wiki.fasterxml.com/JacksonPolymorphicDeserialization

关于说明文件的更多信息，参考下面网站：

https://github.com/metamx/druid/wiki/Realtime

现在，我们将注意力转移到 Druid 侧的实现上。Firehose 是我们要把数据发送给 Druid 实时服务必须要实现的接口。

StormFirehoseFactory 类代码如下：

```
@JsonTypeName("storm")
public class StormFirehoseFactory implements FirehoseFactory {
    private static final StormFirehose FIREHOSE =
    new StormFirehose();
    @JsonCreator
```

```
    public StormFirehoseFactory() {
    }

    @Override
    public Firehose connect() throws IOException {
        return FIREHOSE;
    }

    public static StormFirehose getFirehose(){
        return FIREHOSE;
    }
}
```

这个工厂类的实现非常简单。本例中，我们简单的返回了一个静态 singleton 对象。注意这个对象有 @JsonTypeName 和 @JsonCreator 的注解。前面代码明确了，FirehoseFactory 对象注册的方式是 Jackson。因此，作为 @JsonTypeName 指定的名称必须和说明文件中指定的类型一样。

实现的关键在 StormFirehose 类。在这个类里，有四个关键的方法，我们一个一个来看：hasMore()、nextRow()、commit() 和 sendMessages()。

sendMessages() 方法是 StormFirehose 的入口。它是 Storm 和 Druid 之间有效的切换点。代码如下所示：

```
public synchronized void sendMessages(String partitionId,
                    List<FixMessageDto> messages) {
    BLOCKING_QUEUE =
    new ArrayBlockingQueue<FixMessageDto>(messages.size(),
    false, messages);
    TRANSACTION_ID = partitionId;
    LOG.info("Beginning commit to Druid. [" + messages.size() +
    "] messages, unlocking [START]");
    synchronized (START) {
        START.notify();
    }
    try {
        synchronized (FINISHED) {
        FINISHED.wait();
        }
    } catch (InterruptedException e) {
        LOG.error("Commit to Druid interrupted.");
    }
    LOG.info("Returning control to Storm.");
}
```

这个方法实现为同步的，能避免并发造成的问题。注意它只做一件事情，把消息缓存

拷贝到队列里，并且通知 hasMore() 方法释放这批数据。然后，这个方法会一直阻塞，等待 Druid 完全消费这批数据。

然后，流程进入到 nextRow() 方法，如下所示：

```
@Override
public InputRow nextRow() {
    final Map<String, Object> theMap =
    Maps.newTreeMap(String.CASE_INSENSITIVE_ORDER);
    try {
    FixMessageDto message = null;
    message = BLOCKING_QUEUE.poll();

    if (message != null) {
    LOG.info("[" + message.symbol + "] @ [" +
     message.price + "]");
    theMap.put("symbol", message.symbol);
    theMap.put("price", message.price);
    }

    if (BLOCKING_QUEUE.isEmpty()) {
    STATUS.putInLimbo(TRANSACTION_ID);
    LIMBO_TRANSACTIONS.add(TRANSACTION_ID);
    LOG.info("Batch is fully consumed by Druid. "
    + "Unlocking [FINISH]");
    synchronized (FINISHED) {
        FINISHED.notify();

    }
    }
} catch (Exception e) {
    LOG.error("Error occurred in nextRow.", e);
    System.exit(-1);
}
final LinkedList<String> dimensions =
new LinkedList<String>();
dimensions.add("symbol");
dimensions.add("price");
return new MapBasedInputRow(System.currentTimeMillis(),
                            dimensions, theMap);
}
```

这个方法将消息从队列中拉出。如果不是 null，则把数据添加到一个 map 中，作为 MapBasedInputRow 的一个参数传递给 Druid。如果队列中没有消息了，释放我们前面代码中讲过的 sendMessages() 方法。从 Storm 的角度来看，这批数据已经处理完了。Druid 现在拥有这批数据。然而，从系统的角度看，这些数据处于 limbo 状态，因为 Druid 可能还

没有把这些数据落地到硬盘上。如果硬件故障，我们就有丢失这部分数据的风险。

Druid 然后轮询 hasMore() 方法，如下面代码所示：

```
@Override
public boolean hasMore() {
    if (BLOCKING_QUEUE != null && !BLOCKING_QUEUE.isEmpty())
        return true;
    try {
        synchronized (START) {
        START.wait();
        }
    } catch (InterruptedException e) {
        LOG.error("hasMore() blocking interrupted!");
    }
    return true;
}
```

因为队列是空的，这个方法会阻塞直到 sendMessage() 再次被调用。这时候只剩下一个问题了，commit() 方法。它的实现在下面的代码中展示：

```
@Override
public Runnable commit() {
List<String> limboTransactions = new ArrayList<String>();
LIMBO_TRANSACTIONS.drainTo(limboTransactions);
return new StormCommitRunnable(limboTransactions);
}
```

这个方法返回一个 Runnable 对象，当 Druid 持久化存储消息完成之后调用。虽然 Firehose 对象中其他方法都是单线程调用的，但 Runnable 会在不同的线程中调用，因此它必须是线程安全的。为了达到目的，我们复制 limbo 中的事务到一个独立的列表中，并且将之传递给 Runnable 的构造函数中。你在下面的代码中可以看到，Runnable 做的唯一的事情是将事务移动到 ZooKeeper 的 completed 状态中。

```
public class StormCommitRunnable implements Runnable {
    private List<String> partitionIds = null;

    public StormCommitRunnable(List<String> partitionIds){
        this.partitionIds = partitionIds;
    }

    @Override
    public void run() {
    try {
        StormFirehose.STATUS.complete(partitionIds);
    } catch (Exception e) {
```

```
            Log.error("Could not complete transactions.", e);
        }
    }
}
```

7.4.3 在 ZooKeeper 中实现分片状态

现在我们已经审阅了所有的代码，现在来看看状态是如何持久存储在 ZooKeeper 中的。这样做可以使系统协调分布式处理，尤其是在系统故障产生时。

示例这个实现利用了 ZooKeeper 来持久化存储分片处理状态。ZooKeeper 是另外一个开源项目。要了解详情，参见 http://zookeeper.apache.org/。

ZooKeeper 维护一个树形节点。每个节点有一个对应的路径，和文件系统非常类似。这个实现通过一个叫 Curator 的框架来使用 ZooKeeper。更多信息参考 http://curator.incubator.apache.org/。

当通过 Curator 连接到 ZooKeeper 时，你需要提供一个名字空间。实际上，它就是存储应用程序信息的顶级目录节点。在我们的实现中，名字空间叫做 stormdruid。应用程序在它下面维护了三个路径，用来存储分片数据的状态信息。

路径对应的状态描述和设计如下：

- /stormdruid/current：对应 current 状态，正在处理中。
- /stormdruid/limbo：对应 limbo 状态，处理完，是否落地存储未知。
- /stormdruid/completed：对应 completed 状态，已经提交。

在我们的实现中，对分片状态进行的所有 ZooKeeper 的交互都是通过 DruidPartitionStatus 类执行的。该类的代码如下：

```
public class DruidBatchStatus {
    private static final Logger LOG =
LoggerFactory.getLogger(DruidBatchStatus.class);
    final String COMPLETED_PATH = "completed";
    final String LIMBO_PATH = "limbo";
    final String CURRENT_PATH = "current";
    private CuratorFramework curatorFramework;

    public DruidBatchStatus() {
    try {
curatorFramework =
    CuratorFrameworkFactory.builder()
    .namespace("stormdruid")
```

```
            .connectString("localhost:2181")
            .retryPolicy(new RetryNTimes(1, 1000))
            .connectionTimeoutMs(5000)
                    .build();
                curatorFramework.start();

                if (curatorFramework.checkExists()
            .forPath(COMPLETED_PATH) == null) {
                curatorFramework.create().forPath(COMPLETED_PATH);
                }

            }catch (Exception e) {
                LOG.error("Could not establish connection to Zookeeper",
                        e);
            }
            }

            public boolean isInLimbo(String paritionId) throws Exception {
                return (curatorFramework.checkExists().forPath(LIMBO_PATH
                + "/" + paritionId) != null);
            }
            public void putInLimbo(Long paritionId) throws Exception {
            curatorFramework.inTransaction().
                delete().forPath(CURRENT_PATH + "/" + paritionId)
                .and().create().forPath(LIMBO_PATH + "/" +
                                        paritionId).and().commit();
            }
        }
```

为了节省篇幅，我们只显示了构造函数和 limbo 状态相关的方法。在构造函数中，客户端连接 ZooKeeper 并且建立三个如上文描述的基础路径。然后，提供查询方法来判断一个事务是在处理中（current），还是在 limbo，或者已经处理完毕（completed）。还提供了方法将事务在这些状态中流转。

7.5　执行实现的程序

代码已经就绪，让我们来执行示例！我们通过 FinancialAnalyticsTopology 类的 main 方法启动 topology。为了更好的演示，我们随机生成 0 到 100 的价格（参考前面的 Emitter 类的代码）。

一旦 topology 启动后，你将会看到下列输出：

```
2014-02-16 09:47:15,479-0500 | INFO [Thread-18]
DefaultCoordinator.initializeTransaction(24) | Initializing
Transaction [1615]
2014-02-16 09:47:15,482-0500 | INFO [Thread-22]
DruidState.commit(28) | Committing partition [0] of batch [1615]
2014-02-16 09:47:15,484-0500 | INFO [Thread-22]
StormFirehose.sendMessages(82) | Beginning commit to Druid. [7996]
messages, unlocking [START]
2014-02-16 09:47:15,511-0500 | INFO [chief-stockinfo]
StormFirehose.nextRow(58) | Batch is fully consumed by Druid.
Unlocking [FINISH]
2014-02-16 09:47:15,511-0500 | INFO [Thread-22]
StormFirehose.sendMessages(93) | Returning control to Storm.
2014-02-16 09:47:15,513-0500 | INFO [Thread-18]
DefaultCoordinator.success(30) | Successful Transaction [1615]
```

你可以从多个维度查看处理过程。

使用 ZooKeeper 客户端，可以查看事务处理的状态。看下面列出的代码，它们展示了事务 / 数据批次的标识符以及处理状态：

```
[zk: localhost:2181(CONNECTED) 50] ls /stormdruid/current
[501-0]
[zk: localhost:2181(CONNECTED) 51] ls /stormdruid/limbo
[486-0, 417-0, 421-0, 418-0, 487-0, 485-0, 484-0, 452-0, ...
[zk: localhost:2181(CONNECTED) 82] ls /stormdruid/completed
[zk: localhost:2181(CONNECTED) 52] ls /stormdruid/completed
[59-0, 321-0, 296-0, 357-0, 358-0, 220-0, 355-0,
```

要进行监控和告警，请注意以下事项：

- 如果多批数据处在 current 路径下，需要告警
- 如果 limbo 中的数据批次标识符不连续，或者 limbo 中的标识符在正在处理数据的标识符之后，需要告警

为了清理 ZooKeeper 中的状态，可以执行下述代码：

```
zk: localhost:2181(CONNECTED) 83] rmr /stormdruid
```

可以用 MySql 客户端来监控分段传播情况。使用默认的 schema，你可以从 prod_segments 表中选出分段数：

```
mysql> select * from prod_segments;
```

7.6 检视分析过程

现在，我们等待的时刻终于来了；我们可以使用 Druid 提供的 REST API 来查看一段

时间内的股票价格均值。为了使用 REST API，没有必要运行一个完整的 Druid 集群。只需要一个单独的嵌入式实时节点进行查询数据即可，但每个节点其实都有响应请求的能力，这使得测试非常方便。使用 curl 程序，你可以通过下面命令向实时节点发送查询请求：

```
curl -sX POST "http://localhost:7070/druid/v2/?pretty=true" -H
'content-type: application/json'  -d @storm_query
```

curl 命令的最后一个参数表示一个文件，它包括了 post 请求中 body 的内容。该文件包括以下内容：

```
{
    "queryType": "groupBy",
    "dataSource": "stockinfo",
    "granularity": "minute",
    "dimensions": ["symbol"],
    "aggregations":[
        { "type": "longSum", "fieldName": "orders",
         "name": "cumulativeCount"},
        { "type": "doubleSum", "fieldName": "totalPrice",
         "name": "cumulativePrice" }
    ],
    "postAggregations":[
    {  "type":"arithmetic",
        "name":"avg_price",
        "fn":"/",
        "fields":[ {"type":"fieldAccess","name":"avgprice",
        "fieldName":"cumulativePrice"},
                  {"type":"fieldAccess","name":"numrows",
        "fieldName":"cumulativeCount"}]}
    ],
    "intervals":["2012-10-01T00:00/2020-01-01T00"]
}
```

Druid 中发生了两种聚合操作。一部分聚合操作发生在生成索引时，一部分聚合发生在查询时。发生在索引阶段的聚合在 spec 文件中定义。回忆一下，我们在 spec 文件中定义了两种聚合操作：

```
"aggregators": [
{ "type": "count", "name": "orders"},
   { "type": "doubleSum", "fieldName": "price",
"name": "totalPrice" }
],
```

我们进行聚合的事件具有两个字段：symbol 和 price。前面的聚合发生在建立索引的阶段，引入了两个额外的字段：totalPrice 和 orders。回忆 totalPrice 是一个时间片段内每个

事件价格的累加和。Orders 字段包含了一个时间片段中包括的事件的个数的计数。

然后，在执行一次查询时，Druid 基于 groupBy 语句执行了第二种聚合操作。在我们的查询中，我们以一分钟的粒度根据 symbol 进行分组操作。这种聚合操作引入了两个新的字段：cumulativeCount 和 cumulativePrice。这些字段包含了前面的聚合操作结果的求和值。

最后，我们引入了一个 postaggregation 方法来计算一段时间内的均值。Postaggregation 方法对两个累加字段做除法，结果存到到一个新的 avg_price 字段。

发送的 curl 语句到一个运行的 server 上，会得到以下返回：

```
[ {
  "version" : "v1",
  "timestamp" : "2013-05-15T22:31:00.000Z",
  "event" : {
    "cumulativePrice" : 3464069.0,
    "symbol" : "MSFT",
    "cumulativeCount" : 69114,
    "avg_price" : 50.12108979367422
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-05-15T22:31:00.000Z",
  "event" : {
    "cumulativePrice" : 3515855.0,
    "symbol" : "ORCL",
    "cumulativeCount" : 68961,
    "avg_price" : 50.98323690201708
  }
...
 {
  "version" : "v1",
  "timestamp" : "2013-05-15T22:32:00.000Z",
  "event" : {
    "cumulativePrice" : 1347494.0,
    "symbol" : "ORCL",
    "cumulativeCount" : 26696,
    "avg_price" : 50.47550194785736
  }
}, {
  "version" : "v1",
  "timestamp" : "2013-05-15T22:32:00.000Z",
  "event" : {
    "cumulativePrice" : 707317.0,
    "symbol" : "SPY",
```

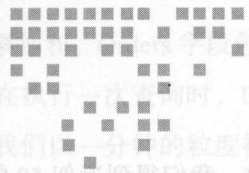
```
    "cumulativeCount" : 13453,
    "avg_price" : 52.576897346316805
  }
} ]
```

因为我们更新了代码来生成 0 到 100 之间的随机数，我们得到近似 50 的平均值也就没什么奇怪的了。

总结

本章中，我们深入理解了 Trident State API。建立了一个 State 和 StateUpdater 接口的自定义实现来替代默认的实现。特别是，我们实现了这些接口来桥接事务型 spout 和非事务型系统 Druid。虽然不能在非事务型存储上确立唯一性语义，但我们可以实现系统异常时的告警机制。表面上看，遇到故障时我们可以使用批处理机制来重建任何不再可靠的聚合片段。

为了进一步研究，在 Storm 和 Druid 之间建立一个幂等的接口是非常有益的。为了实现这种接口，我们可以在 Storm 对应每批数据发布一个单独的分段。因为 Druid 中的分段传播是原子性的，这就可以实现 Druid 中每批数据自动提交的机制。此外，每批数据可以并发的处理，提高了吞吐量。Druid 支持无限扩展的查询类型集合和聚合机制。这是非常强大的功能，Storm 和 Druid 属于强强联合。



Chapter 8 第 8 章

自然语言处理

在实时分析和数据处理需求增长之下，一些人认为 Storm 会最终代替 Hadoop。本章中，我们将看到 Storm 和 Hadoop 实际上是互补的关系。

虽然 Storm 模糊了传统的联机事务处理（On-Line Transactional Processing，OLTP）和联机分析处理（On-Line Analytical Processing，OLAP）之间的界限，它可以处理大量的事务并且可以进行聚合操作和量纲分析（dimensional analysis），通常会与数据仓库结合。它还经常会需要额外的基础设施来执行历史数据分析，支持对全量数据集的临时查询。此外，批处理系统常常用来纠正 OLTP 系统中因故障事件导致的不一致的异常。我们在 Storm-Druid 集成的过程中正好遇到过这个问题。

基于很多原因，批处理系统常常结合实时系统来使用。Hadoop 提供了一个批处理框架。本章中，我们会实现一个体系结构通过批处理支持历史数据和临时数据的分析。

本章介绍以下主题：

- CAP 原理。
- Lambda 体系结构。
- OLTP and OLAP 集成。
- Hadoop 介绍。

8.1　Motivating Lambda 结构

首先，让我们从逻辑上来看一下 Storm-Druid 的集成。Storm，特别是 Trident，可以进行分布式的分析是因为它隔离出了状态转换操作。为了实现这个功能，Storm 为底层持久化存储状态的机制做了一些假定。Storm 假定持久化存储机制具有一致性和可用性。特别是，Storm 假定一旦一个状态转换完成后，新的状态在多个节点上是共享的，一致的并且立即可用。

从 CAP 原理来看，我们知道分布式系统很难同时保证下面三个属性：

- Consistency 一致性：所有节点上的状态一致。
- Availability 可用性：无论成功或者失败系统都可以返回结果。
- Partition Tolerance 分区容错性：在部分系统故障或者失去联系后，整个系统仍然可以提供服务。

越来越多的互联网规模的体系结构集成进来的持久化机制牺牲了一定的一致性，来满足高可用性和分区容错性。通常，这些系统这样做的主要原因是，在一个大型分布式系统中保证事务一致性所需要的协调工作很难实现。而性能和吞吐量是更重要的指标。

Druid 也做了同样的权衡。如果我们看看 Druid 的持久化模型，我们会看到一些不同的步骤，参见图 8-1。

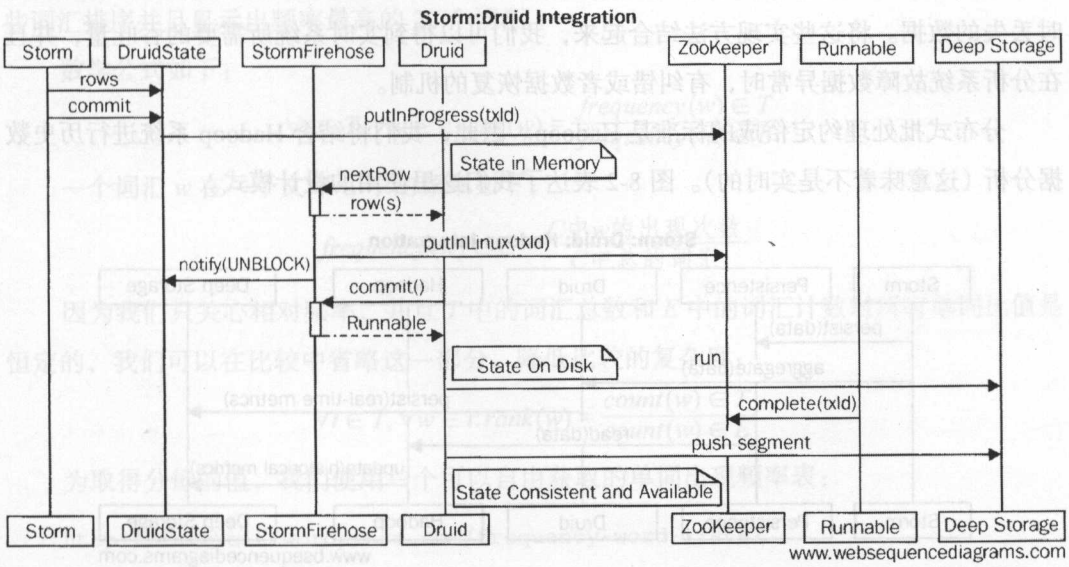


图　8-1

首先，Druid 通过 Firehose 接口消费数据，并且将数据放在内存中。然后，数据持久化存储到硬盘上，并且通过一个 Runnable 接口通知到 Firehose 的实现。最后，数据推送到深度存储器中，这时候这些数据就可以被系统的其他部分使用了。

现在，如果我们考虑如何对非一致性数据进行容错，我们会发现在数据写入到深度存储器之前都是有风险的。如果我们丢失了节点，我们会丢失了该节点上所有从 Storm 拉来的数据分析结果，因为我们已经对这些 tuple 应答确认了。

一个明显的解决方案是在分段数据写入深度存储器之后再在 Storm 中对 tuple 应答确认。这看起来是可行的，但是会在 Storm 和 Druid 之间建立一定的耦合关系。特别是，Storm 每批数据的大小和超时时间都需要和 Druid 里每段数据的大小和分段数据写入深度存储器的事件相匹配。另一个角度讲，事务型处理系统的吞吐量会受到限制，预知绑定的分析系统的吞吐量同样受限。总而言之，这种相互依赖的特性是我们不想要的。

但是，我们还是想使用实时分析并且对部分数据丢失或者系统错误具有一定的容错能力。从这个方面看，上述集成方式是满足需求的。最理想的情况下，我们应该有一个错误纠正和恢复机制。为了实现这种能力，我们将介绍一个离线批处理机制，它在遇到故障事件时能够恢复和纠错数据。

为了达到这个目的，我们首先在数据发送到 Druid 之前将它们持久化存储。我们的批处理系统会从持久化存储机制中离线读取数据。批处理系统可以纠错 / 更新任何实时处理时丢失的数据。将这些实现方法结合起来，我们可以得到实时系统所需要的吞吐量，并且在分析系统故障数据异常时，有纠错或者数据恢复的机制。

分布式批处理约定俗成的标准是 Hadoop。因此，我们将结合 Hadoop 系统进行历史数据分析（这意味着不是实时的）。图 8-2 表达了我们这里使用的设计模式。

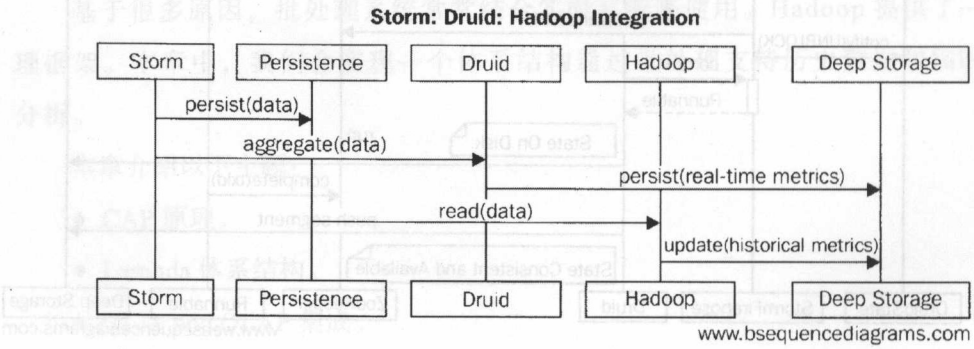


图 8-2

前面的模式展示了如何成功整合 OLTP 和 OLAP 系统，提供一致和完整的实时分析，同时具备高吞吐量、可用性和分片能力，同时还提供了机制说明部分系统故障的原因。

这种实现方式提供了在系统中引入新型分析的能力。因为 Storm-Druid 集成主要解决实时问题，没有简便的方法在系统中引入新型分析。Hadoop 补上了这个短板，因为它是运行在历史数据之上，很容易增加新的维度或者执行额外的聚合操作。

Storm 的最初作者 Nathan Marz 将这种实现方式命名为“Lambda architecture”。

8.2　研究使用场景

现在，让我们将这个设计模式应用在自然语言处理领域（Natural Language Processing，NLP）。这个使用场景中，我们会在 Twitter 中查找特定词组相关的推文（例如“Apple Jobs”）。然后，系统会处理这些推文，尝试找出最相关的词语。使用 Druid 对这些词汇进行聚合，我们会随着时间不同找出最相关词汇的趋势。

让我们更详细地定义这个问题。给出一个待查找的短语 p，使用 Twitter API，我们找到最相关的推文集合 T。对 T 中每条推文 t，我们会计算每个单词出现的次数 w。我们会比较推文中词汇出现的频率和一个示例英文文本 E 中对应词汇出现的频率。然后系统为这些词汇排序并且显示出频率最高的 20 个词汇。

数学公式如下：

$$\forall t \in T, \forall w \in t: rank(w) = \frac{frequency(w) \in T}{frequency(w) \in E}$$

一个词汇 w 在一个文库 C 中的频率如下：

$$frequency(w) \in C = \frac{C\text{中}w\text{的出现次数}}{C\text{中总的词汇}}$$

因为我们只关心相对频率，并且 T 中的词汇总数和 E 中的词汇计数对所有单词比值是恒定的，我们可以在比较中省略这一部分，降低比较的复杂度：

$$\forall t \in T, \forall w \in t: rank(w) = \frac{count(w) \in T}{count(w) \in E}$$

为取得分母的值，我们使用一个可以自由获取的单词出现频率表：

```
http://invokeit.wordpress.com/frequency-word-lists/
```

我们会使用 Storm 来处理 Twitter 查询结果，并且在 tuple 中加入对应的分母的计数

信息。Druid 然后为分子统计出现次数，再使用一个 post-aggregation 来执行实际的相关性计算。

8.3 实现 Lambda architecture

为解决这个应用场景，我们采用一个分布式计算模式，它将实时计算平台（Storm）和一个数据分析引擎（Druid）整合在一起；然后再引入离线批处理机制（Hadoop）来提供历史数据分析的能力。

在上述关注点之外，我们尝试达到的另一个目的是持续可用性和容错性。详细地说，系统可以容忍部分节点，甚至是一个数据中心出现故障。为了达到这种可用性和容错性，我们需要更多关注持久化存储。

在一个运行的系统中，我们可以使用一个分布式的存储机制进行持久化，理想的存储机制支持数据中心间的冗余复制。这样即使极端情况下一个数据中心故障了，整个系统仍然可以在没有数据丢失的情况下恢复。和持久存储交互的时候，客户端会要求一个一致性级别，在事务进行过程中会将数据发送到多个数据中心。

讨论中，假设我们使用 Cassandra 作为持久化存储机制。Cassandra 具有可调节的一致性级别，数据写入时会 EACH_QUORUM 作为一致性配置。这样数据的拷贝会写入所有数据中心。当然这样会导致数据中心之间通信的负担。对不是很核心的应用程序，LOCAL_QUORUM 更适合一些，这会避免数据内部数据中心之间的通信负担，只在同一个数据中心内进行备份。

使用 Cassandra 这类分布式存储引擎的另一个好处是可以专门为离线 / 批处理系统单独设置 Ring/ 集群。Hadoop 可以使用这个 Ring 作为数据输入，允许系统重复使用历史数据而不影响事务的处理。考虑图 8-3 中的体系结构。

在图中，我们有两个数据中心，每个 Cassandra 集群服务于 Storm 的事务处理。这允许 Storm 中实时写入的数据都有一份冗余拷贝，如果使用 EACH_QUORUM 模式会在 tuple 应答确认之前进行复制，或者使用 LOCAL_QUORUM 时在确认后进行复制。此外，我们还有一个虚拟数据中心用来支持离线批处理。Ring 3 是一个 Cassandra 集群和 Ring1 在物理上并列的，但是在 Cassandra 里配置为第二个数据中心。当我们运行一个 Hadoop job 来处理历史数据，我们可以使用 LOCAL_QUORUM。因为本地投票算法只在本地数据中心中获取一致性，Hadoop 读取数据不会影响到事务处理的集群。

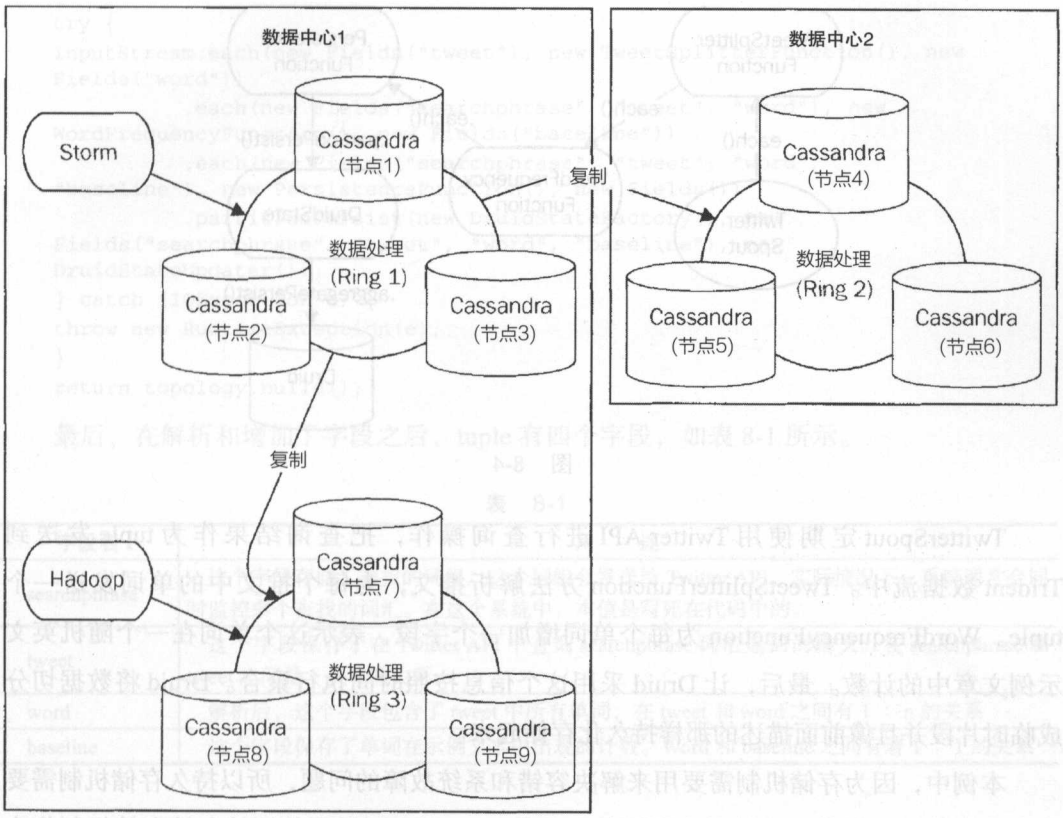


图　8-3

总之这是一种非常好的设计模式，如果你的组织中有专门的数据科学家 / 管理员进行数据分析。通常，这些工作是数据密集型的。将这些分析工作和事务交易系统隔离是非常重要的。

此外，被证明和容错性同样重要的能力是，这个体系结构允许我们在系统中引入数据提取时不具备的新的分析方法。Hadoop 可以使用新的分析配置运行在所有相关历史数据上，增加新的维度或者执行新的聚合。

8.4　为应用场景设计 topology

本例中，我们会再次使用 Trident，并且在前一章节 topology 的基础上构造新的 topology。Trident topology 如图 8-4 所示。

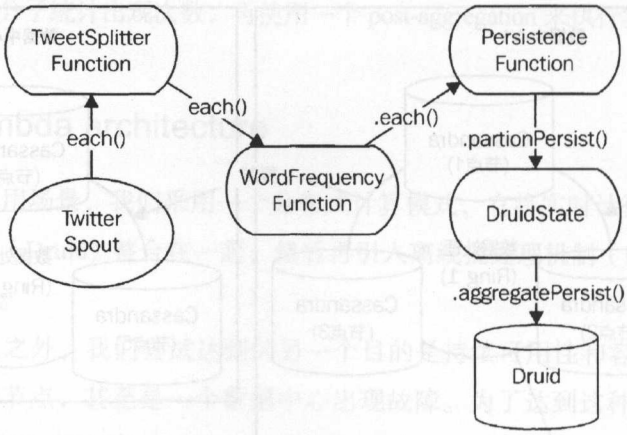


图 8-4

TwitterSpout 定期使用 Twitter API 进行查询操作，把查询结果作为 tuple 发送到 Trident 数据流中。TweetSplitterFunction 方法解析推文，为每个推文中的单词发送一个 tuple。WordFrequencyFunction 为每个单词增加一个字段，表示这个单词在一个随机英文示例文章中的计数。最后，让 Druid 采用这个信息按照时间执行聚合。Druid 将数据切分成临时片段并且像前面描述的那样持久化存储起来。

本例中，因为存储机制需要用来解决容错和系统故障的问题，所以持久存储机制需要分布式存储并且提供一致性和高可用性。此外，Hadoop 还需要使用持久性存储机制作为 map/reduce job 的输入。

因为有可配置调节的一致性以及兼容 Hadoop，Cassandra 是适用于这个设计模式的一个理想的持久性存储机制。因为 Cassandra 和多语言支持的持久性机制其他章节已经介绍过了，这个例子为了简单起见使用本地文件进行存储。

8.5 设计的实现

首先让我们研究一下系统中实时部分，从 spout 开始一直到 Druid 的持久存储。topology 非常简单，与我们前几章实现过的 topology 类似。

下面是 toplogy 中的关键代码：

```
TwitterSpout spout = new TwitterSpout();
Stream inputStream = topology.newStream("nlp", spout);
```

```
try {
inputStream.each(new Fields("tweet"), new TweetSplitterFunction(), new
Fields("word"))
        .each(new Fields("searchphrase", "tweet", "word"), new
WordFrequencyFunction(), new Fields("baseline"))
        .each(new Fields("searchphrase", "tweet", "word",
"baseline"), new PersistenceFunction(), new Fields())
        .partitionPersist(new DruidStateFactory(), new
Fields("searchphrase", "tweet", "word", "baseline"), new
DruidStateUpdater());
} catch (IOException e) {
throw new RuntimeException(e);
}
return topology.build();
```

最后，在解析和增加了字段之后，tuple 有四个字段，如表 8-1 所示。

表　8-1

字段名	用　途
searchphrase	这个字段存储了查找的词组。这个词组会发送给 Twitter API。实际情况下，系统通常会同时监控多个查找的词汇。在这个系统中，本值是写死在代码中的。
tweet	这个字段保存了在 Twitter API 中查询 searchphrase 词汇返回的推文。在 searchphrase 和 tweet 之间是 1 ： n 的关系。
word	解析后，这个字段包含了 tweet 中所有单词，在 tweet 和 word 之间有 1 ： n 的关系
baseline	这个字段保存了单词在示例文本中出现的计数。Word 和 baseline 之间有着 1 ： 1 的关系

8.5.1　TwitterSpout/TweetEmitter

现在来看一下 spout/emitter。本例中，我们使用了 Twitter4J API，并且 Emitter function 就是简单地将 Twitter4J API 和 Storm API 粘合在一起。如前面所示，它使用 Twitter4J 调用 Twitter API 并且将结果作为一批数据发送到 Storm 中。

在一个更复杂的场景下，我们会查询 Twitter Firehose 并且在将数据发送到 Storm 之前先使用一个队列缓存起来。下面是 spout 中 Emitter 部分的关键代码行：

```
query = new Query(SEARCH_PHRASE);
query.setLang("en");
result = twitter.search(query);
...
for (Status status : result.getTweets()) {
    List<Object> tweets = new ArrayList<Object>();
    tweets.add(SEARCH_PHRASE);
    tweets.add(status.getText());
    collector.emit(tweets);
}
```

8.5.2 function

这部分包括 topology 使用的 function。在这个例子中，所有的 funciton 都有特殊的功能（例如持久化数据），或者在 tuple 中增加字段。

TweetSplitterFunction

推文经过的第一个 function 是 TweetSplitterFunction。这个 function 简单地解析推文并且为推文中每个单词发送一个 tuple。下面是 funciton 的代码：

```
@Override
public void execute(TridentTuple tuple, TridentCollector collector) {
String tweet = (String) tuple.getValue(0);
LOG.debug("SPLITTING TWEET [" + tweet + "]");
Pattern p = Pattern.compile("[a-zA-Z]+");
Matcher m = p.matcher(tweet);
List<String> result = new ArrayList<String>();
    while (m.find()) {
        String word = m.group();
        if (word.length() > 0) {
          List<Object> newTuple = new ArrayList<Object>();
          newTuple.add(word);
          collector.emit(newTuple);
        }
    }
}
```

在更复杂的 NLP 系统中，这个 function 不仅仅只按照空白符对推文进行单词分割。NLP 系统很可能会解析推文，对其进行分词，将相关的词汇关联在一起。虽然，即时通信消息和推文通常缺乏训练分词器需用到的传统语法结构，但是系统可以使用元素关联方法进行分词，比如单词之间的距离。这种系统中，会使用 n-gram 语言模型的频率代替单词频率，n-gram 是由多个单词组成的。

从这里可以了解到 n-grams，https://books.google.com/ngrams

WordFrequencyFunction

现在开始看 `WordFrequencyFunction`。这个 function 在 tuple 中添加 baseline 计数。这个数字是该单词在一段随机样本文字中出现的次数。

这个 funciton 关键代码如下：

```
public static final long DEFAULT BASELINE = 10000;
```

```
private Map<String, Long> wordLikelihoods =
new HashMap<String, Long>();

public WordFrequencyFunction() throws IOException {
File file = new File("src/main/resources/en.txt");
BufferedReader br = new BufferedReader(new FileReader(file));
String line;
while ((line = br.readLine()) != null) {
String[] pair = line.split(" ");
   long baseline = Long.parseLong(pair[1]);
   LOG.debug("[" + pair[0] + "]=>[" + baseline + "]");
   wordLikelihoods.put(pair[0].toLowerCase(), baseline);
   i++;
}
br.close();
}

@Override
public void execute(TridentTuple tuple,
TridentCollector collector) {
String word = (String) tuple.getValue(2);
Long baseline = this.getLikelihood(word);
List<Object> newTuple = new ArrayList<Object>();
newTuple.add(baseline);
collector.emit(newTuple);
}

public long getLikelihood(String word){
Long baseline = this.wordLikelihoods.get(word);
if (baseline == null)
return DEFAULT_BASELINE;
else
   return baseline;
}
```

代码中的构造函数将单词计数加载到内存中。En.txt 文件的格式如下：

```
you 4621939
the 3957465
i 3476773
to 2873389
...
of 1531878
that 1323823
in 1295198
is 1242191
me 1208959
what 1071825
```

每行包括一个单词以及这个单词出现的频率。再次强调的是，因为我们只关心相对计数，我们不用在意样本中的单词总数。

这个 funciton 中的 execute 很简单，在 tuple 中添加了一个 baseline 计数。然而，如果我们研究这个从 HashMap 类中获取单词计数方法时，有一个 DEFAULT_BASELINE 变量。当系统遇到一个在词库中找不到的单词时，会使用这个默认值作为计数值返回。

因为 Twitter 的消息会包括很多缩略语、首字母缩写词以及其他一些在标准文本中找不到的词汇，DEFAULT_BASELINE 就成为了一个很重要的配置参数。有时候，出现一次的单词很重要且唯一，因为属于 searchphrase 字段。而有的情况下比较异常，因为样本词库和对象词库不一样。

理想情况下，目标文本的原始 baseline 计数最好从同样的文本源统计得而来。本例中，最理想的就是从整个 Twitter Firehose 中算出单词和 n-gram 计数。

PersistenceFunction

我们不会深入 Cassandra 多集群部署的细节。本例为了简单起见，使用本地文件存储。PersistenceFunction 代码如下：

```
@Override
public void execute(TridentTuple tuple,
   TridentCollector collector) {
writeToLog(tuple);
collector.emit(tuple);
}

synchronized public void writeToLog(TridentTuple tuple) {
DateTime dt = new DateTime();
DateTimeFormatter fmt = ISODateTimeFormat.dateTime();
StringBuffer sb = new StringBuffer("{ ");
sb.append(String.format("\"utcdt\":\"%s\",", fmt.print(dt)));
sb.append(String.format("\"searchphrase\":\"%s\",", tuple.
getValue(0)));
sb.append(String.format("\"word\":\"%s\",", tuple.getValue(2)));
sb.append(String.format("\"baseline\":%s", tuple.getValue(3)));
sb.append("}");
BufferedWriter bw;
try {
bw = new BufferedWriter(new FileWriter("nlp.json", true));
bw.write(sb.toString());
   bw.newLine();
   bw.close();
```

```
} catch (IOException e) {
    throw new RuntimeException(e);
}
}
```

在上面的代码中，function 简单地将 tuple 按照原始格式持久化存储，Druid 在 Hadoop 索引 job 中会去消费这些数据。这段代码效率低下，因为每次写入都会打开关闭文件。作为选择，我们可以实现额外的 StateFactory 和 State 对象来持久化存储数据；因为这只是一个示例，我们可以容忍这种低效的文件访问。

此外，注意我们在这里生成了一个时间戳，没有随着 tuple 一起发送。理想情况下，我们应该生成一个时间戳并添加到 tuple 里，可以被下游的 Druid 用作时序分片。这个例子里，我们接受这个与实际不符的地方。

提示　即使这个 function 没有给 tuple 添加字段，它还是要重新发送 tuple。因为 function 也扮演了 filter 的角色，每个 function 有责任声明哪些 tuple 需要发送到下游。

这个 function 把下列这些行写到了 nlp.json 文件中：

```
{ "utcdt":"2013-08-25T14:47:38.883-04:00","searchphrase":"apple jobs",
"word":"his","baseline":279134}
{ "utcdt":"2013-08-25T14:47:38.884-04:00","searchphrase":"apple jobs",
"word":"annual","baseline":839}
{ "utcdt":"2013-08-25T14:47:38.885-04:00","searchphrase":"apple jobs",
"word":"salary","baseline":1603}
{ "utcdt":"2013-08-25T14:47:38.886-04:00","searchphrase":"apple jobs",
"word":"from","baseline":285711}
{ "utcdt":"2013-08-25T14:47:38.886-04:00","searchphrase":"apple jobs",
"word":"Apple","baseline":10000}
```

8.6　检视分析逻辑

Druid 集成和上一章的用法一样。简要回顾一下，这个集成有 StateFactory、StateUpdater、State 的实现组成。State 实现通过 StormFirehoseFactory 和 StormFirehose 实现相互通信与 Druid 交互。这个实现中的核心是 StormFirehose 的实现，将 tuple 映射到 Druid 读取的一行行的数据。这个方法的代码如下：

```
@Override
public InputRow nextRow() {
```

```
    final Map<String, Object> theMap =
Maps.newTreeMap(String.CASE_INSENSITIVE_ORDER);
try {
TridentTuple tuple = null;
    tuple = BLOCKING_QUEUE.poll();
    if (tuple != null) {
String phrase = (String) tuple.getValue(0);
        String word = (String) tuple.getValue(2);
        Long baseline = (Long) tuple.getValue(3);
        theMap.put("searchphrase", phrase);
        theMap.put("word", word);
        theMap.put("baseline", baseline);
}

    if (BLOCKING_QUEUE.isEmpty()) {
        STATUS.putInLimbo(TRANSACTION_ID);
        LIMBO_TRANSACTIONS.add(TRANSACTION_ID);
        LOG.info("Batch is fully consumed by Druid. Unlocking
[FINISH]");
        synchronized (FINISHED) {
            FINISHED.notify();
        }
    }
} catch (Exception e) {
LOG.error("Error occurred in nextRow.", e);
}
final LinkedList<String> dimensions = new LinkedList<String>();
dimensions.add("searchphrase");
dimensions.add("word");
return new MapBasedInputRow(System.currentTimeMillis(),
dimensions, theMap);
}
```

仔细看这个方法，有两个关键的数据结构：theMap 和 dimensions。第一个包含每行的数据值，第二个包括每行的维度，Druid 会用这个值来进行聚合运算，并且决定了对这些数据可以执行哪些查询操作。本例中，我们使用 searchphrase 和 word 作为维度。这允许我们在查询中执行统计和分组，稍后会看到。

首先，我们查看 Druid 提取数据的配置。配置的大部分可以复用前面一章讲的嵌入式实时服务中的配置。数据段会发送到 Cassandra 进行深度存储，MySQL 会存储数据段的元数据。

下面是 runtime.properties 文件中关键配置参数：

```
druid.pusher.cassandra=true
druid.pusher.cassandra.host=localhost:9160
```

```
druid.pusher.cassandra.keyspace=druid
druid.zk.service.host=localhost
druid.zk.paths.base=/druid
druid.host=127.0.0.1
druid.database.segmentTable=prod_segments
druid.database.user=druid
druid.database.password=druid
druid.database.connectURI=jdbc:mysql://localhost:3306/druid
druid.zk.paths.discoveryPath=/druid/discoveryPath
druid.realtime.specFile=./src/main/resources/realtime.spec
druid.port=7272
druid.request.logging.dir=/tmp/druid/realtime/log
```

配置文件指向了 realtime.spec 文件，它指定实时分析服务执行分析的详情。下面是本例中用到的 realtime.spec 文件：

```
[{
    "schema": {
        "dataSource": "nlp",
        "aggregators": [
            { "type": "count", "name": "wordcount" },
            { "type": "max", "fieldName": "baseline",
name" : "maxbaseline" }
        ],
        "indexGranularity": "minute",
        "shardSpec": {"type": "none"}
    },

    "config": {
        "maxRowsInMemory": 50000,
        "intermediatePersistPeriod": "PT30s"
    },

    "firehose": {
        "type": "storm",
        "sleepUsec": 100000,
        "maxGeneratedRows": 5000000,
        "seed": 0,
        "nTokens": 255,
        "nPerSleep": 3
    },

    "plumber": {
        "type": "realtime",
        "windowPeriod": "PT10s",
        "segmentGranularity": "minute",
        "basePersistDirectory": "/tmp/nlp/basePersist"
    }
}]
```

在时间粒度之外，这个文件中还规定了聚合器。它告诉 Druid 如何在数据的行之间聚合数据。没有聚合器，Druid 就没有办法压缩数据。这个应用场景中，有两个聚合器：wordcount 和 maxbaseline。

wordcount 字段当每行的字段有相同值时，累加对应维度的计数。参考 StormFirehose 的实现，两个维度是 searchphrase 和 word。这样，Druid 可以压缩行数据，增加一个新的字段叫做 wordcount，保存了一个时间粒度中，该单词出现的总次数。

Maxbaseline 字段保存了每个单词对应的 baseline。实际上，这个值对每行都是一样的。我们简单地使用了 max 函数作为一个方便的函数将该值传递给聚合操作，在使用查询时可以用到。

现在，让我们来进行查询。下面的查询用来提取出最相关的单词：

```
{
     "queryType": "groupBy",
     "dataSource": "nlp",
     "granularity": "minute",
     "dimensions": ["searchphrase", "word"],
     "aggregations":[
        { "type": "longSum", "fieldName":"wordcount",
"name": "totalcount"},
        { "type": "max", "fieldName":"maxbaseline",
"name": "totalbaseline"}
     ],
     "postAggregations": [{
       "type": "arithmetic",
       "name": "relevance",
       "fn": "/",
       "fields": [
            { "type": "fieldAccess", "fieldName": "totalcount" },
            { "type": "fieldAccess", "fieldName": "totalbaseline" }
       ]
     }],
     "intervals":["2012-10-01T00:00/2020-01-01T00"]
 }
```

查询需要和 realtime.spec 文件对应起来。在查询底部，我们指定了感兴趣的时间区间。在文件顶部，我们指定了感兴趣的维度，后面跟着的是指定 Druid 按照时间粒度压缩列所用的聚合操作。这个例子中，聚合操作刚好和我们对数据进行实时索引时使用的聚合操作相同。

特别的，我们引入了 totalcount 字段，包含了 wordcount 字段的总和。这个量包括了

该 word 和 searchphrase 组合出现实例的总次数。此外，我们对 baseline 做同样的事情并将对应的值传到下游。

最后，在这个查询中，我们包括了一个后聚合操作，它会将聚合结果合并为一个对应的值。后聚合将推文中的计数除以基线出现的次数。

下列代码是一个简单的 Ruby 文件处理查询结果并且返回 top20 的单词：

```
...
url="http://localhost:7272/druid/v2/?pretty=true"
response = RestClient.post url, File.read("realtime_query"), :accept
=> :json, :content_type => 'appplication/json'
#puts(response)
result = JSON.parse(response.to_s)

word_relevance = {}
result.each do |slice|
  event = slice['event']
  word_relevance[event['word']]=event['relevance']
end

count = 0
word_relevance.sort_by {|k,v| v}.reverse.each do |word, relevance|
  puts("#{word}->#{relevance}")
  count=count+1
  if(count == 20) then
    break
  end
end
```

注意，我们用来访问服务的 URL 是嵌入式实时服务绑定的端口。生产环境中，查询是通过一个 broker 节点进行的。

执行脚本的结果如下所示：

```
claiming->31.789579158316634
apple->27.325982081323225
purchase->20.985449735449734
Jobs->20.618
Steve->17.446
shares->14.802238805970148
random->13.480033984706882
creation->12.7524115755627
Apple->12.688
acts->8.82582081246522
prevent->8.702687877125618
farmer->8.640522875816993
developed->8.62642740619902
```

```
jobs->8.524986566362172
bottles->8.30523560209424
technology->7.535137701804368
current->7.21418826739427
empire->6.924050632911392
```

如果你改变了捕获数据的维度或度量，要确保删除实时服务缓存数据的本地文件夹。否则，实时服务可能重新读取不含有新维度或度量的老数据，这些数据无法满足查询；此外查询还可能因为 Druid 无法找到所需要的度量或者维度而执行失败。

批处理 / 历史数据分析

现在，我们把注意力转到批处理机制上。我们用 Hadoop 来实现这个功能。虽然 Hadoop 的完整介绍超出了本章的范围，我们还是会在下节中特别安装 Druid 的过程中间简单介绍一下。

Hadoop 提供了两个关键组件：一个分布式文件系统和一个分布式处理框架。分布式文件系统叫做 Hadoop Distributed Filesystem（HDFS）。分布式处理框架就是大家都熟知的 MapReduce。因为我们选择利用 Cassandra 作为假想的系统存储机制，我们不需要使用 HDFS。但是我们会使用 Hadoop 中的 MapReduce 部分进行历史数据的分布式处理。

在我们的例子中，会执行本地的 Hadoop job 读取 PersistenceFunction 写入的文件。这个例子中会使用伴随着 Hadoop job 的 Druid。

8.7 Hadoop

在载入数据之前，需要对 MapReduce 简单介绍一下。虽然 Druid 已经打包为便于执行 MapReduce job 来适应历史数据，一般而言，大型分布式系统都会需要自定义的 job 来分析整个数据集。

8.7.1 MapReduce 概览

MapReduce 是一个框架，将数据处理分成了两步：一个 map 步骤和一个 reduce 步骤。在 map 步骤中，对所有输入数据执行一个 function，每个执行一次。map function 的

每次应用返回一组 tuple，每个数据包含一个 key 和一个 value。相同 key 的 tuple 通过执行 reduce function 进行合并。reduce function 发送另外一组 tuple，通常是由特定 key 的值合并的结果。

MapReduce 官方的“Hello Word”例子是单词计数。给定一组包含单词的文档，对每个单词出现的次数计数（有趣的是，和我们 NLP 的例子非常相似）。

下面是一个 Ruby 函数用来表达单词计数中的 map 和 reduce 函数。map 函数如下所示：

```
def map(doc)
   result = []
doc.split(' ').each do |word|
result << [word, 1]
   end
   return result
end
```

对下面的输入，map 函数产出如下的输出：

```
map("the quick fox jumped over the dog over and over again")
 => [["the", 1], ["quick", 1], ["fox", 1], ["jumped", 1], ["over",
1], ["the", 1], ["dog", 1], ["over", 1], ["and", 1], ["over", 1],
["again", 1]]
```

对应的 reduce 函数如下面代码段所示：

```
def reduce(key, values)
   sum = values.inject { |sum, x| sum + x }
   return [key, sum]
end
```

MapReduce 函数然后按照 key 将 value 进行分组，像下面这样将值传递给 reduce 函数，得到的结果是全局的单词计数：

```
reduce("over", [1,1,1])
 => ["over", 3]
```

8.7.2　Druid 安装

用 Hadoop 作为后端，让我们了解一下 Druid 的安装过程。为了让 Druid 能够从 Hadoop job 中消费数据，我们需要启动 Master 和 Compute 节点（也叫做 Historical 节点）。为了做到这点，我们建立一个目录结构，用 Druid 的自包含 job 作为根目录，子目录包括 Master 和 Compute 服务的配置。

这个目录结构像下面这样：

```
druid/druid-indexing-hadoop-0.5.39-SNAPSHOT.jar
druid/druid-services-0.5.39-SNAPSHOT-selfcontained.jar
druid/config/compute/runtime.properties
druid/config/master/runtime.properties
druid/batchConfig.json
```

Master 和 Compute 节点的运行时属性基本和前面讲的实时节点一样，只有微小差异。都包括缓存数据段的设置，如下所示：

```
# Path on local FS for storage of segments;
# dir will be created if needed
druid.paths.indexCache=/tmp/druid/indexCache
# Path on local FS for storage of segment metadata;
# dir will be created if needed
druid.paths.segmentInfoCache=/tmp/druid/segmentInfoCache
```

还需要注意的是，如果你要把 Master 和 Compute 节点运行在同一个机器上，需要改变它们绑定的端口避免冲突：

```
druid.port=8082
```

Druid 将所有服务组件和它们的依赖打包到一个自包含的 Jar 文件中。按照下面命令使用这个 Jar 文件，就可以启动 Master 和 Compute 服务。

使用下面命令来启动 Compute 节点：

```
java -Xmx256m -Duser.timezone=UTC -Dfile.encoding=UTF-8 \
-classpath ./druid-services-0.5.39-SNAPSHOT-selfcontained.jar:config/
compute \
com.metamx.druid.http.ComputeMain
```

使用下面命令启动 Master 节点：

```
java -Xmx256m -Duser.timezone=UTC -Dfile.encoding=UTF-8 \
-classpath ./druid-services-0.5.39-SNAPSHOT-selfcontained.jar:config/
compute \
com.metamx.druid.http.ComputeMain
```

一旦两个节点都在运行了，我们可以开始给 Hadoop job 载入数据了。

HadoopDruidIndexer

服务启动之后，现在可以查看 Druid MapReduce job 的内部细节了。HadoopDruidIndexer 函数使用一个和 realtime.spec 文件类似的 JSON 配置文件。

这个文件用来指定 Hadoop job 启动的命令行，代码如下所示：

```
java -Xmx256m -Duser.timezone=UTC -Dfile.encoding=UTF-8 \
```

```
-Ddruid.realtime.specFile=realtime.spec -classpath druid-services-
0.5.39-SNAPSHOT-selfcontained.jar:druid-indexing-hadoop-0.5.39-
SNAPSHOT.jar \
com.metamx.druid.indexer.HadoopDruidIndexerMain batchConfig.json
```

下面是本例中用到的 batchConfig.json:

```
{
  "dataSource": "historical",
  "timestampColumn": "utcdt",
  "timestampFormat": "iso",
  "dataSpec": {
    "format": "json",
    "dimensions": ["searchphrase", "word"]
  },
  "granularitySpec": {
    "type":"uniform",
    "intervals":["2013-08-21T19/PT1H"],
    "gran":"hour"
  },
  "pathSpec": { "type": "static",
                "paths": "/tmp/nlp.json" },
  "rollupSpec": {
            "aggs": [ { "type": "count", "name": "wordcount" },
                      { "type": "max", "fieldName": "baseline",
                                       "name" : "maxbaseline" } ],
      "rollupGranularity": "minute"},
      "workingPath": "/tmp/working_path",
  "segmentOutputPath": "/tmp/segments",
  "leaveIntermediate": "false",
  "partitionsSpec": {
    "targetPartitionSize": 5000000
  },
  "updaterJobSpec": {
    "type":"db",
    "connectURI":"jdbc:mysql://localhost:3306/druid",
    "user":"druid",
    "password":"druid",
    "segmentTable":"prod_segments"
  }
}
```

这个配置的大部分看起来都很眼熟。两个有特别意义的字段是 pathSpec 和 rollupSpec。pathSpec 字段包含了 PersistenceFunction 写文件的位置。rollupSpec 文件包含了事务处理中的聚合操作，与 realtime.spec 文件中的一样。

此外注意，我们还指定了时间戳的列和格式，需要和输出在持久化存储的文件中的字

段相匹配：

```
{ "utcdt":"2013-08-25T14:47:38.883-04:00","searchphrase":"apple jobs",
"word":"his","baseline":279134}
{ "utcdt":"2013-08-25T14:47:38.884-04:00","searchphrase":"apple jobs",
"word":"annual","baseline":839}
{ "utcdt":"2013-08-25T14:47:38.885-04:00","searchphrase":"apple jobs",
"word":"salary","baseline":1603}
{ "utcdt":"2013-08-25T14:47:38.886-04:00","searchphrase":"apple jobs",
"word":"from","baseline":285711}
{ "utcdt":"2013-08-25T14:47:38.886-04:00","searchphrase":"apple jobs",
"word":"Apple","baseline":10000}
```

HadoopDruidIndexer 函数载入了前面的配置文件并且执行 map/reduce 来建立索引。如果更仔细看这个 job，我们可以看到它执行了哪些函数。

Hadoop job 使用 Hadoop job 类启动。Druid 运行了两个 job 对数据进行索引，我们主要看 IndexGeneratorJob。在 IndexGeneratorJob 中，Druid 通过下面代码行配置 job：

```
job.setInputFormatClass(TextInputFormat.class);
job.setMapperClass(IndexGeneratorMapper.class);
job.setMapOutputValueClass(Text.class);
...
job.setReducerClass(IndexGeneratorReducer.class);
job.setOutputKeyClass(BytesWritable.class);
job.setOutputValueClass(Text.class);
job.setOutputFormatClass(IndexGeneratorOutputFormat.class);
FileOutputFormat.setOutputPath(job,config.makeIntermediatePath());
config.addInputPaths(job);
config.intoConfiguration(job);
...
job.setJarByClass(IndexGeneratorJob.class);
job.submit();
```

前面的属性基本上都是为 Hadoop job 设定的。指定了处理阶段中每个阶段输入输出的类，以及实现 Mapper 和 Reducer 接口的类。

完整的 Hadoop job 配置的说明，在下面的 URL 查看：http://hadoop.apache.org/docs/r0.18.3/mapred_tutorial.html#Job+Configuration

job 的配置文件还指定了输入路径，表示从哪些文件或者数据源读取数据进行处理。调用 config.addInputPaths，Druid 从 pathSpec 字段添加文件到 Hadoop 的配置文件中进行处理，如下列代码所示：

```
  @Override
  public Job addInputPaths(HadoopDruidIndexerConfig config,
Job job) throws IOException {
```

```
    log.info("Adding paths[%s]", paths);
    FileInputFormat.addInputPaths(job, paths);
    return job;
  }
```

可以看到明显的问题，Druid 只支持 FileInputFormat 的实例。作为练习，可以尝试加强 DruidHadoopIndexer 函数的功能支持直接从 Cassandra 里读取数据，作为理想结构的扩展。

回顾 job 的配置，Druid 使用的 Mapper 类是 IndexGeneratorMapper 类，使用的 Reducer 类是 IndexGeneratorReducer 类。

让我们先看一下 IndexGeneratorMapper 类中的 map 函数。IndexGeneratorMapper 类实际上是 HadoopDruidIndexerMapper 的子类，包含了一个 map 方法的实现，它将委托 IndexGeneratorMapper 类发射实际的数据，下面的代码中会看到：

在 HadoopDruidIndexerMapper 中，map 方法的实现如下：

```
@Override
protected void map(LongWritable key, Text value, Context context
  ) throws IOException, InterruptedException
  {
    try {
      final InputRow inputRow;
      try {
        inputRow = parser.parse(value.toString());
      }
      catch (IllegalArgumentException e) {
        if (config.isIgnoreInvalidRows()) {
          context.getCounter(HadoopDruidIndexerConfig.
IndexJobCounters.INVALID_ROW_COUNTER).increment(1);
          return; // we're ignoring this invalid row
        } else {
          throw e;
        }
      }
      if(config.getGranularitySpec().bucketInterval(new
DateTime(inputRow.getTimestampFromEpoch())).isPresent()) {
        innerMap(inputRow, value, context);
      }
    }
    catch (RuntimeException e) {
      throw new RE(e, "Failure on row[%s]", value);
    }
  }
```

我们可以看到，子类的 map 方法处理每行数据逻辑是，会把无法解析的数据标记为非法，并且检查它是否有执行 map 操作所必须的数据。要强调的是，由超类来保证每行数

据都有一个时间戳。在 abstract 方法中调用 innerMap 时可以看到，map 因为需要将数据按照时间片段分割（也就是 bucket），所以必须用到时间戳，如下所示：

```
@Override
protected void innerMap(InputRow inputRow,
        Text text,
        Context context
    ) throws IOException, InterruptedException{

 // Group by bucket, sort by timestamp
final Optional<Bucket> bucket = getConfig().getBucket(inputRow);

if (!bucket.isPresent()) {
throw new ISE("WTF?! No bucket found for row: %s", inputRow);
}

context.write(new SortableBytes(
              bucket.get().toGroupKey(),
              Longs.toByteArray(inputRow.getTimestampFromEpoch())
          ).toBytesWritable(),text);
}
```

这个方法中关键的一行，也是任何基于 Hadoop 的 map 函数中关键的一行是调用 context.write 从 map 函数中发射 tuple。这个例子中 map 方法发射一个字段，类型是 SortableBytes，表示 bucket 的度量，从输入源读取的文本作为字段的值进行存储。在 map 步骤执行完毕的时候，我们已经解析完了文件，构造好了 bucket，并且将划分好的数据按照时间戳排序存入这些桶。然后每个 bucket 会由 reduce 方法处理，如下所示：

```
@Override
protected void reduce(BytesWritable key, Iterable<Text> values,
final Context context
    ) throws IOException, InterruptedException{
SortableBytes keyBytes = SortableBytes.fromBytesWritable(key);
Bucket bucket = Bucket.fromGroupKey(keyBytes.getGroupKey()).lhs;

final Interval interval =
config.getGranularitySpec().bucketInterval(bucket.time).get();
final DataRollupSpec rollupSpec = config.getRollupSpec();
final AggregatorFactory[] aggs = rollupSpec.getAggs().toArray(
        new AggregatorFactory[rollupSpec.getAggs().size()]);

IncrementalIndex index = makeIncrementalIndex(bucket, aggs);
...
for (final Text value : values) {
context.progress();
```

```
    final InputRow inputRow =
index.getSpatialDimensionRowFormatter()
.formatRow(parser.parse(value.toString()));
        allDimensionNames.addAll(inputRow.getDimensions());
      ...
IndexMerger.persist(index, interval, file,
index = makeIncrementalIndex(bucket, aggs);
      ...
    }
  ...
);
...
serializeOutIndex(context, bucket, mergedBase,
 Lists.newArrayList(allDimensionNames));
...
}
```

可以看到，reduce 方法包括分析的内容。它基于聚合的结果构造索引，聚合操作是按照 roll up 配置以及批处理配置文件中指定的维度进行的。方法中的最后一行将数据段写入硬盘。

最后，当你运行 DruidHadoopIndexer 类时，会看到类似下面的代码段：

```
2013-08-28 04:07:46,405 INFO [main] org.apache.hadoop.mapred.JobClient
-   Map-Reduce Framework
2013-08-28 04:07:46,405 INFO [main] org.apache.hadoop.mapred.JobClient
-     Reduce input groups=1
2013-08-28 04:07:46,405 INFO [main] org.apache.hadoop.mapred.JobClient
-     Combine output records=0
2013-08-28 04:07:46,405 INFO [main] org.apache.hadoop.mapred.JobClient
-     Map input records=201363
2013-08-28 04:07:46,405 INFO [main] org.apache.hadoop.mapred.JobClient
-     Reduce shuffle bytes=0
2013-08-28 04:07:46,406 INFO [main] org.apache.hadoop.mapred.JobClient
-     Reduce output records=0
2013-08-28 04:07:46,406 INFO [main] org.apache.hadoop.mapred.JobClient
-     Spilled Records=402726
2013-08-28 04:07:46,406 INFO [main] org.apache.hadoop.mapred.JobClient
-     Map output bytes=27064165
2013-08-28 04:07:46,406 INFO [main] org.apache.hadoop.mapred.JobClient
-     Combine input records=0
2013-08-28 04:07:46,406 INFO [main] org.apache.hadoop.mapred.JobClient
-     Map output records=201363
2013-08-28 04:07:46,406 INFO [main] org.apache.hadoop.mapred.JobClient
-     Reduce input records=201363
2013-08-28 04:07:46,433 INFO [main] com.metamx.druid.indexer.
IndexGeneratorJob - Adding segment historical_2013-08-
28T04:00:00.000Z_2013-08-28T05:00:00.000Z_2013-08-28T04:07:32.243Z to
the list of published segments
2013-08-28 04:07:46,708 INFO [main] com.metamx.druid.indexer.
DbUpdaterJob - Published historical_2013-08-28T04:00:00.000Z_2013-08-
```

```
28T05:00:00.000Z_2013-08-28T04:07:32.243Z
2013-08-28 04:07:46,754 INFO [main] com.metamx.druid.indexer.
IndexGeneratorJob - Adding segment historical_2013-08-
28T04:00:00.000Z_2013-08-28T05:00:00.000Z_2013-08-28T04:07:32.243Z to
the list of published segments
2013-08-28 04:07:46,755 INFO [main] com.metamx.druid.indexer.
HadoopDruidIndexerJob - Deleting path[/tmp/working_path/
historical/2013-08-28T040732.243Z]
```

注意，添加的数据段叫做 historical。要查询历史 / 批处理机制载入的数据，需要更新查询中历史数据源并且使用一个 Compute 节点绑定的端口。所有需要的准备就位后，你会接收到实时系统返回的聚合结果；一个例子如下所示：

```
{
  "version" : "v1",
  "timestamp" : "2013-08-28T04:06:00.000Z",
  "event" : {
    "totalcount" : 171,
    "totalbaseline" : 28719.0,
    "searchphrase" : "apple jobs",
    "relevance" : 0.005954246317768724,
    "word" : "working"
  }
}
```

现在，如果我们设置 Hadoop job 定期执行，历史索引会迟于实时索引，但是会持续更新索引，进行纠错和解决系统故障导致的数据异常。

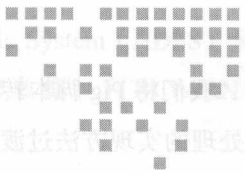
总结

本章中，我们了解了将批处理机制和实时处理引擎（比如 Storm）结合起来，提供更完整和强健的整体解决方案。

我们研究了一种 Lambda 体系结构的实现方法。这种方法使用实时分析功能，并支持通过批处理系统追朔的纠正实时分析的结果。此外，我们还了解了如何配置一个多数据中心的体系结构，从而将离线批处理和事务系统分离开来，同时通过分布式存储提供持续可用性以及容错能力。

本章还介绍了 Hadoop，使用 Druid 的实现作为一个例子。

下一章中，我们会使用一个现成的批处理流程，它利用了 Pig 和 Hadoop，并且示例如何将它转化为实时系统。与此同时，我们还示例如何使用 Storm-YARN 将 Storm 部署在 Hadoop 的体系结构上。



第 9 章 Chapter 9

在 Hadoop 上部署 Storm 进行广告分析

在前面两章中，我们了解了如何将 Storm 和实时分析系统集成起来，然后扩展了该实现，同时给实时处理集成了批处理功能。在本章中，我们会反方向进行探索。

我们会研究用来计算广告投放效率的批处理系统。我们假设这个系统是构建在 Hadoop 上，会将它转换为一个实时系统。

我们利用 Yahoo 开源的 Storm-YARN 项目来实现。Storm-YARN 项目允许利用 YARN 来部署和运行 Storm 集群。在 Hadoop 上运行 Storm 允许企业优化运营成本，并且在实时和批处理系统上能够利用同样的基础设施。

本章主要包括以下主题：

- Pig 简介。
- YARN（Hadoop v2 资源管理）。
- 使用 Storm-YARN 部署 Storm。

9.1 应用场景

我们的使用场景中，会处理一个广告推送的日志来判断哪些是最有效的推送。批处理机制会使用 Pig 脚本处理一个大的非格式化文件。Pig 是一个高层语言，允许用户执行数据事务和分析。它与 SQL 类似，会被编译成 map/reduce job 通常是发布和执行在 Hadoop

基础上。

在本章中，我们将 Pig 脚本转化为 topology 并且使用 Storm-YARN 部署 topology。这允许我们从批处理的实现方法过渡到可以提取和反馈实时事件的实现方法（例如，点击了 banner 上的广告）。

在广告业中，曝光是指一个广告事件将要广告的内容展示给用户，无论是否吸引了用户点击。在我们的分析中，我们会跟踪每一次曝光，并且使用一个字段标记用户是否点击了这个广告。

无格式的文本文件包括四个字段，如表 9-1 所示。

表 9-1

字　段	描　述
cookie	浏览器带上来的唯一标识符。在系统中我们使用这个字段来表示不同的用户
campaign	这个唯一标识符表示一个特定的广告内容集合
product	这是做成广告的产品的名字
click-thru	这是一个布尔字段表示用户是否点击了广告，true 表示点击了，否则是 false

典型的，广告商为了宣传产品举行广告活动。一个广告活动会包括一系列相关的内容。我们想计算出每个产品最有效的广告活动。

基于这个目的，我们通过计算不同用户的点击次数占整体曝光次数的比例，得出一个广告活动的效果。我们会发送如表 9-2 所示的报告。

表 9-2

产　品	广告活动	去重点击量	曝光量
X	Y	107	252

曝光量就是一个产品的一个广告活动曝光的总量。我们不计算去重后的曝光量，因为我们给同一个用户多次展示同一个广告可能会吸引到一次点击。因为我们更可能根据每次曝光进行付费，我们会使用总曝光次数来计算吸引有兴趣的用户的成本。有兴趣的用户就是点击广告的用户。

9.2 确定体系结构

我们在前一章接触了 Hadoop，但是我们主要是为了了解 Hadoop 中的 map/reduce 机

制。在本章中，我们会反过来主要介绍 Hadoop File System（HDFS）和 Yet Another Resource Negotiator（YARN）。我们会利用 HDFS 存储数据，利用 YARN 来部署运行 topology 的 Storm 框架。

当前 Hadoop 框架的组件化允许分布式系统使用它进行资源管理。在 Hadoop 1.0 中，资源管理是嵌入在 MapReduce 框架中的，如图 9-1 所示。

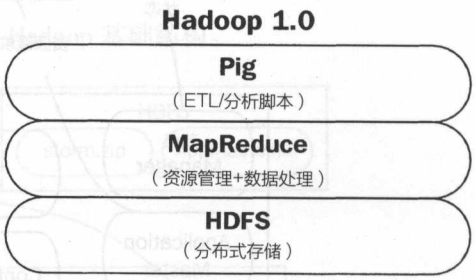


图　9-1

Hadoop 2.0 将资源管理分离到了 YARN 中，允许其他分布式处理系统运行在 Hadoop 结构管理之下的资源上。在我们的例子中，允许我们在 YARN 上运行 Storm，如图 9-2 所示。

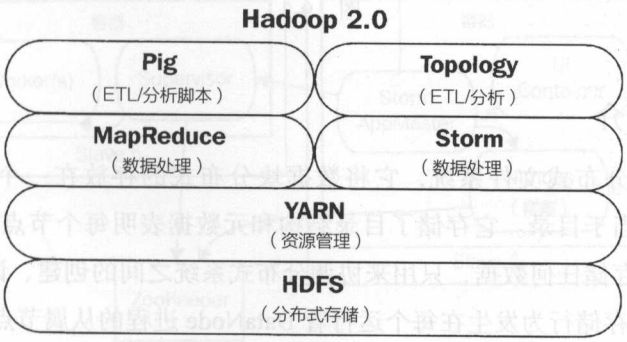


图　9-2

图中 Storm 充当了和 MapReduce 同样的功能。它提供了一个分布式计算的框架。在这个特殊的使用场景中，我们使用 Pig 脚本来执行我们想在数据上执行的 ETL（Extract-Transform-Load）和分析。然后会将这些脚本转化为 Storm toplogy 执行同样的功能，并研究这个转换中引入的难点。

为了更好地理解这种转换，得再来了解一下 Hadoop 集群中的节点和这些节点运行的目的。假设我们有一个集群如图 9-3 所示。

图中展示了两个组件 / 子系统。第一个是 YARN，它是 Hadoop 2.0 中引入的新资源管理层。第二个是 HDFS。让我们首先来研究一下 HDFS，因为它自 Hadoop 1.0 起变化不大。

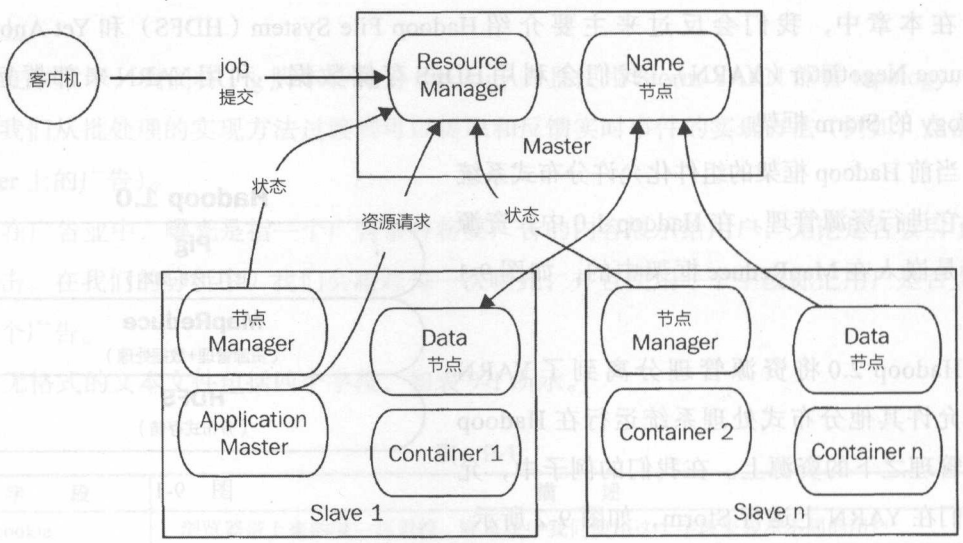


图 9-3

9.2.1 HDFS 简介

HDFS 是一个分布式文件系统，它将数据块分布式的存放在一个从属节点的集合中。NameNode 相当于目录。它存储了目录结构和元数据表明每个节点存储了哪些信息。NameNode 本身不存储任何数据，只用来协调分布式系统之间的创建、读取、更新和操作（CRUD）的操作。存储行为发生在每个运行着 DataNode 进程的从属节点上。DataNode 进程是系统中完成工作的部分。它们相互通信完成数据的负载均衡、冗余备份、移动和复制。由这些节点对客户端的 CRUD 操作进行应答和反馈。

9.2.2 YARN 简介

YARN 是一种资源管理系统，监控每个节点的负载，协调在集群中的从属节点中分配新 job。ResouceManager 收集 NodeManager 中的状态信息，ResourceManager 还提供服务接收客户端提交的 job。

YARN 中的另一个抽象是 ApplicationMaster 的概念。ApplicationMaster 管理某个特定应用的资源和分配的容器，与 ResouceManager 协商来分配资源。一旦分配了资源，ApplicationMaster 和 NodeManager 协商来实例化容器（containers）。实际执行工作的进程运行在容器中。

ApplicationMaster 是一个处理框架相关的库。Storm-YARN 提供了 ApplicationMaster 在 YARN 上启动运行 Storm 进程的功能。HDFS 将 ApplicationMaster 分布在多处，如同 Storm 框架本身一样。目前，Storm-YARN 期望一个外部的 ZooKeeper。应用部署时，Nimbus 启动并且连接 ZooKeeper。

图 9-4 描述了通过 Storm-YARN 运行 Storm 的 Hadoop 基础结构。

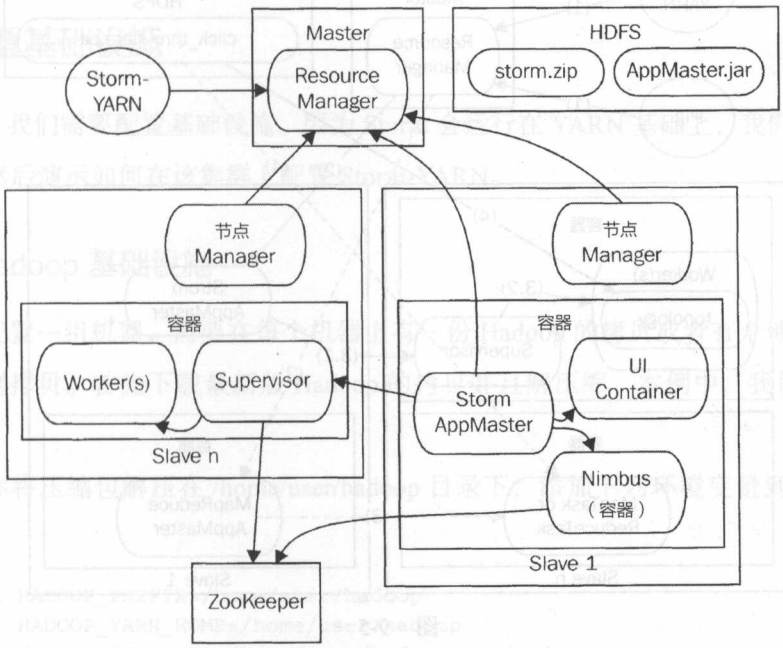


图　9-4

图中 YARN 用来部署 Storm 应用程序框架。在启动时，Storm Application Master 运行在一个 YARN 容器中。这时候，它会建立一个 Storm Nimbus 和 Storm UI 的实例。

之后，Storm-YARN 在一些独立的 YARN 容器中启动 supervisor。每个 supervisor 进程都可以在容器中产生多个 worker。

Application Master 和 Storm 框架都通过 HDFS 实现分布式结构。Storm-YARN 提供了命令行工具来启动 Storm 集群，启动 supervisor，以及修改 Storm 配置来部署 topology。在本章后面我们会了解这些工具。

为了完善体系结构图，我们需要分别描述批处理和实时处理机制：Pig 和 Storm topology。我们还需要描述具体的数据。

通常在 Storm 集群中会采用如 Kafaka 这类队列机制承担队列的工作。为了简化起见，我们将数据存入 HDFS。图 9-5 描述了我们应用场景中用到的 Pig、Storm、YARN 和 HDFS，为了看起来更清晰，忽略了其下层的设施。为了充分实现将 Pig 转化为 Storm 的价值，我们将 topology 从使用 Kafka 切换到使用 HDFS，如图 9-5 所示。

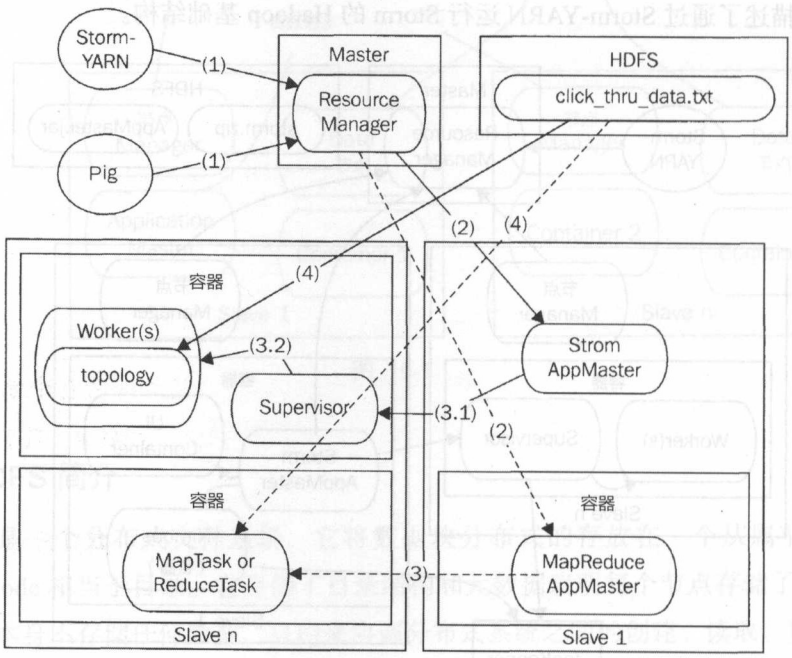


图 9-5

在图中，我们的数据存储在 HDFS 中。虚线表示批处理分析的过程，实线表示实时系统。在每个系统中，会包含的步骤如表 9-2 所示。

表 9-2

步　骤	目　标	Pig 等价操作	Storm-Yarn 等价操作
1	部署处理框架	MapReduce Application Master 部署并启动	Storm-YARN 启动 Application Master 并且分布式发布 Storm 框架
2	启动特定的分析	Pig 脚本编译成 MapReduce jobs，并且提交	Topology 发布到集群
3	预定资源	Map 和 Reduce 任务在 YARN 容器中创建	Supervisor 以及它所带的 worker 被实例化
4	分析程序从存储中读取数据并且执行分析	Pig 从 HDFS 中读取数据	Storm 通常从 Kafka 队列中读取数据。在本例中 topology 直接从非格式化文件中读取数据

Pig 和 Trident 还有一个相似点。Pig 脚本编译成 MapReduce jobs，Trident 编译成 Storm topology。

更多关于 Storm-YARN 项目的信息，访问下面的 URL：

https://github.com/yahoo/storm-yarn

9.3　配置基础设施

首先，我们需要配置基础设施。因为 Storm 会运行在 YARN 基础上，我们首先需要配置 YARN 然后演示如何在该集群上配置 Storm-YARN。

9.3.1　Hadoop 基础设施

为了配置一组机器，需要在每个机器上有一份 Hadoop 的拷贝或者有个地方可以供这些机器复制拷贝。首先下载最新版 Hadoop 的拷贝并且解压缩。本例中，我们使用 2.1.0-beta 版本。

假设你将压缩包解压在 /home/user/hadoop 目录下，添加下列环境变量到集群中的每个节点：

```
export HADOOP_PREFIX=/home/user/hadoop
export HADOOP_YARN_HOME=/home/user/hadoop
export HADOOP_CONF_DIR=/home/user/hadoop/etc/Hadoop
```

将 YARN 添加到可执行路径中：

```
export PATH=$PATH:$HADOOP_YARN_HOME/bin
```

所有的 Hadoop 配置文件都放在 $HADOOP_CONF_DIR 中。本例中三个关键的配置文件是 core-site.xml，yarn-site.xml，和 hdfs-site.xml。

本例中，我们假设有一个叫做 master 的 Master 节点，有四个从属节点叫做 slave01-04。

执行下列命令测试 YARN 配置是否正确：

```
$ yarn version
You should see output similar to the following:
Hadoop 2.1.0-beta
Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1514472
```

```
Compiled by hortonmu on 2013-08-15T20:48Z
Compiled with protoc 2.5.0
From source with checksum 8d753df8229fd48437b976c5c77e80a
This command was run using /Users/bone/tools/hadoop-2.1.0-beta/share/
hadoop/common/hadoop-common-2.1.0-beta.jar
```

9.3.2 配置 HDFS

依照体系结构图，配置 HDFS 需要启动 NameNode 并且将它连接到一个或者多个 DataNode 上。

配置 NameNode

为了启动 NameNode，需要指定一个主机名和端口名。使用下面的标签在 core-site.xml 文件中配置主机名和端口名。

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://master:8020</value>
    </property>
</configuration>
```

此外，还要配置 NameNode 存储元数据的位置。这项配置存储在 hdfs-site.xml 文件中，在 dfs.name.dir 变量下。

为了简化例子，我们会禁用分布式文件系统中的安全选项。我们将 dfs.permissions 设置为 False 来实现这一点。编辑完这些地方后，HDFS 配置文件看起来应该像下面代码片段这样：

```
<configuration>
    <property>
        <name>dfs.name.dir</name>
        <value>/home/user/hadoop/name/data</value>
    </property>
    <property>
        <name>dfs.permissions</name>
        <value>false</value>
    </property>
</configuration>
```

启动 NameNode 前的最后一步是格式化分布式文件系统。执行下面的命令：

```
hdfs namenode -format <cluster_name>
```

最后，已经准备好启动 NameNode 了，执行下列命令来启动：

```
$HADOOP_PREFIX/sbin/hadoop-daemon.sh --config $HADOOP_CONF_DIR --script
hdfs start namenode
```

启动命令的最后一行会指定将 log 存在哪里：

```
starting namenode, logging to /home/user/hadoop/logs/hadoop-master.
hmsonline.com.out
```

提示　不考虑消息，日志实际上存在另外一个文件中，名字一样，但是后缀是 log 而不是 out。而且，保证配置中声明日志存放位置的目录是存在的；否则，会在 log 文件中接收到下面的错误信息：

```
org.apache.hadoop.hdfs.server.common.
InconsistentFSStateException: Directory /home/user/
hadoop-2.1.0-beta/name/data is in an inconsistent
state: storage directory does not exist or is not
accessible.
```

通过下列代码确认 NameNode 已经启动：

```
boneill@master:~-> jps
30080 NameNode
```

此外，还可以通过 web 浏览器访问 UI 界面。默认的，UI 服务绑定在端口 50070 上。浏览 http://master:50070，你会看到的截图如图 9-6 所示。

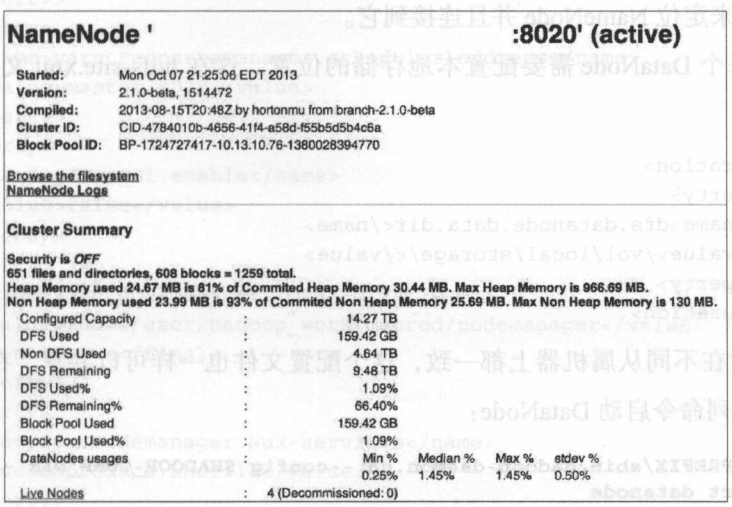
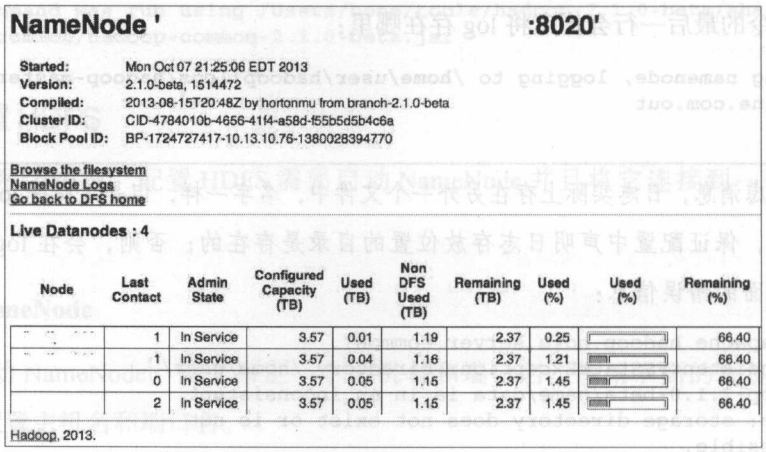


图　9-6

点击 Live Node 链接，会显示当前哪些节点是可用的，以及每个节点上的空间分配，如图 9-7 所示。



NameNode ' **:8020'**

Started: Mon Oct 07 21:25:06 EDT 2013
Version: 2.1.0-beta, 1514472
Compiled: 2013-08-15T20:48Z by hortonmu from branch-2.1.0-beta
Cluster ID: CID-4784010b-4656-41f4-a58d-f55b5d5b4c6a
Block Pool ID: BP-1724727417-10.13.10.76-1380028394770

Browse the filesystem
NameNode Logs
Go back to DFS home

Live Datanodes : 4

Node	Last Contact	Admin State	Configured Capacity (TB)	Used (TB)	Non DFS Used (TB)	Remaining (TB)	Used (%)	Used (%)	Remaining (%)
	1	In Service	3.57	0.01	1.19	2.37	0.25		66.40
	1	In Service	3.57	0.04	1.16	2.37	1.21		66.40
	0	In Service	3.57	0.05	1.15	2.37	1.45		66.40
	2	In Service	3.57	0.05	1.15	2.37	1.45		66.40

Hadoop, 2013.

图 9-7

最后，从主页上，还可以点击 Browse the filesystem 来查看文件系统。

配置 DataNode

通常，在集群的节点之间共享配核心配置非常容易。DataNode 会使用文件 core-site.xml 中的配置来定位 NameNode 并且连接到它。

此外，每个 DataNode 需要配置本地存储的位置。这在 hdfs-site.xml 文件中的下列元素中配置：

```
<configuration>
   <property>
       <name>dfs.datanode.data.dir</name>
       <value>/vol/local/storage/</value>
   </property>
</configuration>
```

如果配置在不同从属机器上都一致，这个配置文件也一样可以共享。在一个集合中，你可以通过下列命令启动 DataNode：

```
$HADOOP_PREFIX/sbin/hadoop-daemon.sh --config $HADOOP_CONF_DIR --script
hdfs start datanode
```

再一次，使用 jps 命令确认 DateNode 是否运行，并监控日志文件的错误。稍等片刻，

DataNode 会出现 NameNode 前台的 Live Nodes 中。

配置 YARN

当 HDFS 启动并且运行后，现在将注意力转回 YARN。类似 HDFS 所做的，我们首先启动 ResourceManager，然后运行 NodeManager 将从属节点添加进来。

配置 ResourceManager

ResourceManager 有多个组成部分，每部分都作为一个服务器需要一个主机和端口来绑定运行。所有的服务器都在 yarn-site.xml 文件中配置。

本例中，我们会使用下述 YARN 配置：

```
<configuration>
   <property>
       <name>yarn.resourcemanager.address</name>
       <value>master:8022</value>
   </property>
   <property>
       <name>yarn.resourcemanager.admin.address</name>
       <value>master:8033</value>
   </property>
   <property>
       <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>master:8025</value>
   </property>
   <property>
       <name>yarn.resourcemanager.scheduler.address</name>
       <value>master:8030</value>
   </property>
   <property>
       <name>yarn.acl.enable</name>
       <value>false</value>
   </property>
   <property>
       <name>yarn.nodemanager.local-dirs</name>
       <value>/home/user/hadoop_work/mapred/nodemanager</value>
       <final>true</final>
   </property>
   <property>
     <name>yarn.nodemanager.aux-services</name>
     <value>mapreduce.shuffle</value>
   </property>
</configuration>
```

上面配置文件中前四个变量指定了组件的主机名和端口。将 yarn.acl.enable 变量设置为 False 可以禁用 YARN 集群中的安全选项。yarn.nodemanager.local-dirs 变量指定了 YARN 将数据放在本地文件系统的哪个位置。

最后，yarn.nodemanager.aux-services 变量会在 NodeManager 运行时启动一个辅助服务来支持 MapReduce job。因为我们的 Pig 脚本编译成了 MapReduce job，它们依赖于这个选项。

和 NameNode 类似，通过下列命令启动 ResourceManager：

```
$HADOOP_YARN_HOME/sbin/yarn-daemon.sh --config $HADOOP_CONF_DIR start
resourcemanager
```

再一次，通过 jps 命令检查进程是否已经启动，查看 log 看看是否有异常，然后可以浏览默认绑定在 8088 端口上的 UI 界面。UI 如图 9-8 所示。

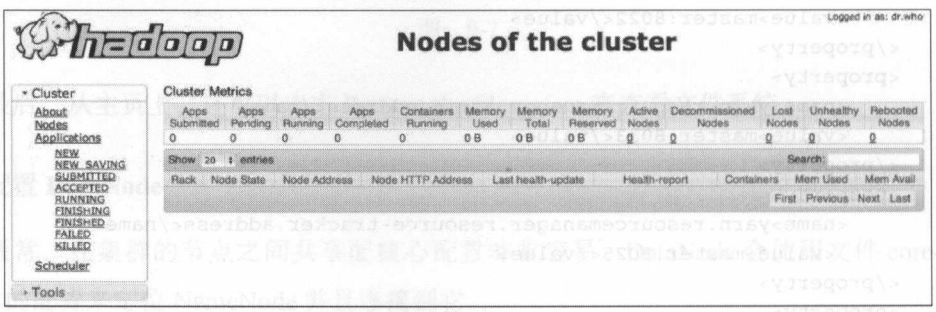


图 9-8

配置 NodeManager

NodeManager 也使用同样的配置文件（yarn-site.xml）来配置各自的服务器。因此，集群中各节点共享这个文件是安全的。

通过下列命令启动 NodeManager：

```
$HADOOP_YARN_HOME/sbin/yarn-daemon.sh --config $HADOOP_CONF_DIR start
nodemanager
```

在所有的 NodeManagers 注册到 ResourceManager 之后，你会在 ResourceManager 的 UI 上通过点击 Node 看到这些节点，如图 9-9 所示。

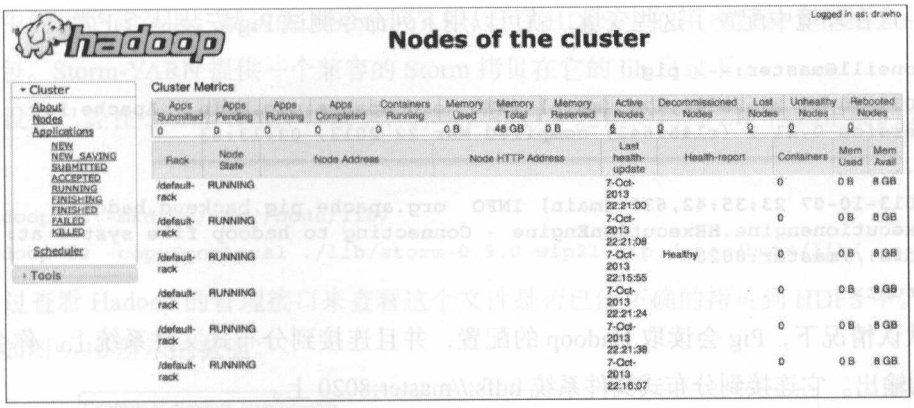


图　9-9

9.4　部署分析程序

Hadoop 就位后，我们现在可以集中精力来分析分布式处理框架。

9.4.1　以 Pig 为基础执行批处理分析

我们研究的第一个分布式处理框架是 Pig。Pig 是一个数据分析的框架。他允许用户使用简单的高级语言来表达分析内容。这些脚本会被编译成 MapReduce job。

虽然 Pig 可以从不同的系统（例如，S3）中读取数据，本例中会使用 HDFS 作为存储机制。因此我们分析的第一步是将数据拷贝到 HDFS 中。

我们执行下面的 Hadoop 命令实现这一点：

```
hadoop fs -mkdir /user/bone/temp
hadoop fs -copyFromLocal click_thru_data.txt /user/bone/temp/
```

上面的命令为数据建立了一个目录并且将广告点击流数据拷贝到这个目录下。

为了在这些数据上执行 Pig 脚本，我们需要安装 Pig。我们简单的下载 Pig 并且将它解压在 Hadoop 配置指定的机器上即可。在这个例子中，我们使用版本 0.11.1。

和 Hadoop 类似，系统中添加下列环境变量：

```
export PIG_CLASSPATH=/home/user/hadoop/etc/hadoop
export PIG_HOME=/home/user/pig
export PATH=PATH:$HOME/bin:$PIG_HOME/bin:$HADOOP_YARN_HOME/bin
```

PIG_CLASSPATH 变量告诉 Pig 在哪里找到 Hadoop。

一旦在环境中配置了这些变量，就可以用下列命令测试 Pig 安装是否正常：

```
boneill@master:~-> pig
2013-10-07 23:35:41,179 [main] INFO  org.apache.pig.Main - Apache Pig
version 0.11.1 (r1459641) compiled Mar 22 2013, 02:13:53
...
2013-10-07 23:35:42,639 [main] INFO  org.apache.pig.backend.hadoop.
executionengine.HExecutionEngine - Connecting to hadoop file system at:
hdfs://master:8020
grunt>
```

默认情况下，Pig 会读取 Hadoop 的配置，并且连接到分布式文件系统上。你会看到上述的输出。它连接到分布式文件系统 hdfs://master:8020 上。

通过 Pig，你可以和普通文件系统一样和 HDFS 进行交互。例如，可以像下面代码段一样使用 ls 和 cat 命令：

```
grunt> ls /user/bone/temp/
hdfs://master:8020/user/bone/temp/click_thru_data.txt<r 3>      157

grunt> cat /user/bone/temp/click_thru_data.txt
boneill campaign7 productX true
lisalis campaign10 productX false
boneill campaign6 productX true
owen campaign6 productX false
collin campaign7 productY true
maya campaign8 productY true
boneill campaign7 productX true
owen campaign6 productX true
olive campaign6 productX false
maryanne campaign7 productY true
dennis campaign7 productY true
patrick campaign7 productX false
charity campaign10 productY false
drago campaign7 productY false
```

9.4.2 在 Storm-YARN 基础上执行实时分析

现在，我们已经有了可以执行批处理的基础设施，让我们利用同样的基础设施来实现实时处理。Storm-YARN 让 Storm 可以很方便地重用 Hadoop 基础设施。

因为 Storm-YARN 是一个新的项目，最好从源代码编译，并且按照 README 文件中的指引进行打包，参见下述 URL：

https://github.com/yahoo/storm-yarn

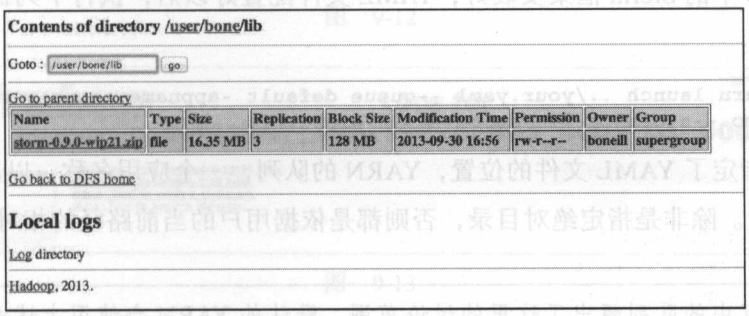
在构建打包后，需要将 Storm 框架拷贝到 HDFS 上。这允许 Storm-YARN 将框架部署

到集群中的每个节点上。默认的，Storm-YARN 会在 HDFS 上的用户目录下查找 Storm 库的 ZIP 包。Storm-YARN 提供一个兼容的 Storm 拷贝在它的 lib 目录下。

假设你已经在 Storm-YARN 目录下，你可以执行下述命令拷贝 ZIP 包到正确的 HDFS 目录下：

```
hadoop fs -mkdir /user/bone/lib/
hadoop fs -copyFromLocal ./lib/storm-0.9.0-wip21.zip /user/bone/lib/
```

通过查看 Hadoop 的管理接口来查看这个文件是否已经正确的拷贝到 HDFS 中。你可以看到如图 9-10 所示的截面。


Contents of directory /user/bone/lib

Goto : /user/bone/lib go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
storm-0.9.0-wip21.zip	file	16.35 MB	3	128 MB	2013-09-30 16:56	rw-r--r--	boneill	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop, 2013.

图　9-10

当 Storm 框架已经存入 HDFS 以后，下一步是为 Storm-YARN 配置本地 YAML 文件。Storm-YAML 使用的 YAML 文件对 Storm-YAML 和 Storm 是公用的。YAML 文件中 Storm 特定的配置参数会被传递给 Storm。

一个 YAML 例子如下面的代码段所示：

```
master.host: "master"
master.thrift.port: 9000
master.initial-num-supervisors: 2
master.container.priority: 0
master.container.size-mb: 5120
master.heartbeat.interval.millis: 1000
master.timeout.secs: 1000
yarn.report.wait.millis: 10000
nimbusui.startup.ms: 10000
ui.port: 7070

storm.messaging.transport: "backtype.storm.messaging.netty.Context"
storm.messaging.netty.buffer_size: 1048576
storm.messaging.netty.max_retries: 100
storm.messaging.netty.min_wait_ms: 1000
storm.messaging.netty.max_wait_ms: 5000
```

```
storm.zookeeper.servers:
    - "zkhost"
```

很多参数是一目了然的。然而，特别注意最后一个参数，它指定了 ZooKeeper 主机的位置。虽然这个选项不总是必须的，当前 Storm-YARN 假设你有一个外部预设好的 ZooKeeper

监控 Storm-YARN 是否继续需要一个预设好的 ZooKeeper 实例，查看下列连接的信息：https://github.com/yahoo/storm-yarn/issues/22。

当 HDFS 中的 Storm 框架安装好，YAML 文件配置好以后，执行下列命令在 YARM 上启动 Storm：

```
storm-yarn launch ../your.yaml --queue default -appname storm-yarn-2.1.0-
deta-demo --stormZip lib/storm-0.9.0-wip21.zip
```

命令中指定了 YAML 文件的位置，YARN 的队列，一个应用名称，以及一个 Storm ZIP 包的位置。除非是指定绝对目录，否则都是依据用户的当前路径的相对路径。

YARN 中的队列超出了这里的讨论范围，默认的 YARN 会使用上述命令配置一个默认的队列。如果你将 Storm 运行在一个预先已经存在的集群上，检查 YARN 配置中的 capacity-scheduler.xml 文件来确认可用的队列名称。

执行了上面的命令行之后，你会在 YARN 的管理界面上看到部署的应用程序，如图 9-11 所示。

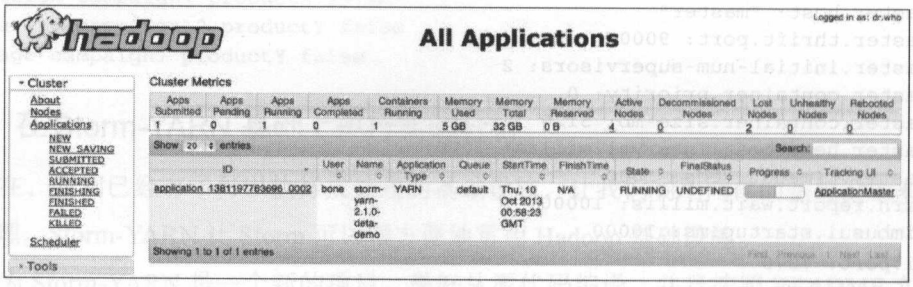


图 9-11

点击应用程序会显示应用程序的主节点部署在哪里。查看 node 的值找到 Application Master。图 9-12 显示了 Storm UI 的位置。

再向下段保护一层，你会看到 Storm 的日志文件，如图 9-13 所示。

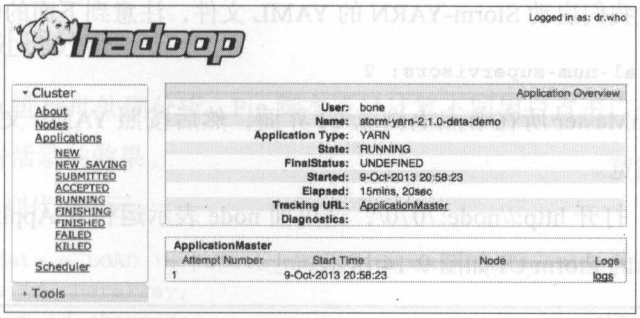


图　9-12

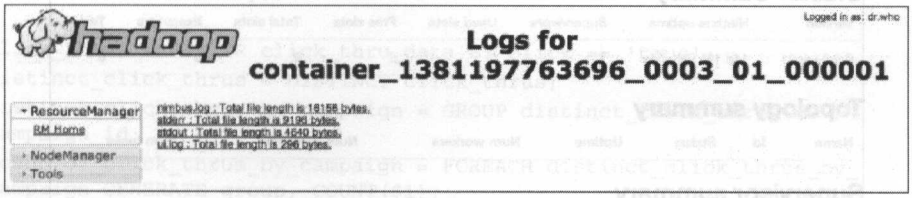


图　9-13

没什么问题的话，日志会显示 Nimbus 和 UI 启动成功。查看标准输出流，你会单刀 Storm-YARN 启动了 supervisor：

```
13/10/09 21:40:10 INFO yarn.StormAMRMClient: Use NMClient to launch
supervisors in container.
13/10/09 21:40:10 INFO impl.ContainerManagementProtocolProxy: Opening
proxy : slave05:35847
13/10/09 21:40:12 INFO yarn.StormAMRMClient: Supervisor
log: http://slave05:8042/node/containerlogs/
container_1381197763696_0004_01_000002/boneill/supervisor.log
13/10/09 21:40:14 INFO yarn.MasterServer: HB: Received allocated
containers (1) 13/10/09 21:40:14 INFO yarn.MasterServer: HB:
Supervisors are to run, so queueing (1) containers...
13/10/09 21:40:14 INFO yarn.MasterServer: LAUNCHER: Taking container
with id (container_1381197763696_0004_01_000004) from the queue.
13/10/09 21:40:14 INFO yarn.MasterServer: LAUNCHER:
Supervisors are to run, so launching container id
(container_1381197763696_0004_01_000004)
13/10/09 21:40:16 INFO yarn.StormAMRMClient: Use NMClient to
launch supervisors in container.  13/10/09 21:40:16 INFO impl.
ContainerManagementProtocolProxy: Opening proxy : dlwolfpack02.
hmsonline.com:35125
13/10/09 21:40:16 INFO yarn.StormAMRMClient: Supervisor
log: http://slave02:8042/node/containerlogs/
container_1381197763696_0004_01_000004/boneill/supervisor.log
```

上面输出的关键行着重标记出来了。如果你浏览这些 URL，会看到不同 Supervisor 实例的日志。回头看我们启动 Storm-YARN 的 YAML 文件，注意到下面的值：

```
master.initial-num-supervisors: 2
```

在 ApplicationMaster 所在节点上浏览 UI 界面，然后按照 YAML 文件中指定的 UI 启动绑定的端口来浏览。

在浏览器中，打开 http://node:7070/，这里面 node 表示运行着 ApplicationMaster 的节点。你会看到熟悉的 Storm UI 如图 9-14 所示。

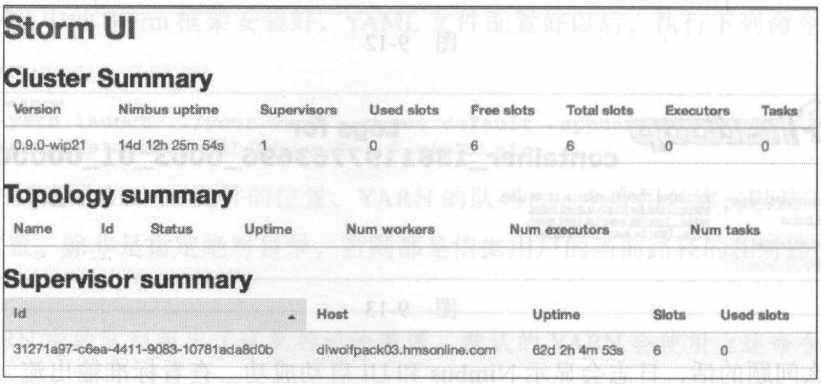


图 9-14

现在基础设施已经可以使用了。要结束在 YARN 上部署的 Storm，可以执行下列命令：

```
./storm-yarn shutdown -appId application_1381197763696_0002
```

在前面的语句中，appId 参数对应分配给 Storm-YARN 的 appId，可以在 Hadoop 的管理界面上看到。

提示 Storm-YARN 会使用本地 Hadoop 配置来确定 Hadoop 的主节点。如果启动一个非 Hadoop 集群中的机器，需要配置该机器的 Hadoop 环境变量和配置文件。特别是，要启动 ResourceManager。因此，需要 yarn-site.xml 中配置下面的变量：

yarn.resourcemanager.address

9.5 执行分析

批处理和实时处理基础设施都就位以后，我们可以集中精力到分析上。首先，看一下

Pig 中的处理，然后会将 Pig 脚本转化为 Storm 的 topology。

9.5.1　执行批处理分析

我们使用 Pig 进行批处理分析。Pig 脚本通过计算不同用户点击广告在总曝光数中的比例，来计算广告活动的效果。

Pig 脚本如下列代码所示：

```
click_thru_data = LOAD '../click_thru_data.txt' using PigStorage(' ')
  AS (cookie_id:chararray,
      campaign_id:chararray,
      product_id:chararray,
      click:chararray);

click_thrus = FILTER click_thru_data BY click == 'true';
distinct_click_thrus = DISTINCT click_thrus;
distinct_click_thrus_by_campaign = GROUP distinct_click_thrus BY
campaign_id;
count_of_click_thrus_by_campaign = FOREACH distinct_click_thrus_by_
campaign GENERATE group, COUNT($1);
-- dump count_of_click_thrus_by_campaign;

impressions_by_campaign = GROUP click_thru_data BY campaign_id;
count_of_impressions_by_campaign = FOREACH impressions_by_campaign
GENERATE group, COUNT($1);
-- dump count_of_impressions_by_campaign;

joined_data = JOIN count_of_impressions_by_campaign BY $0 LEFT OUTER,
count_of_click_thrus_by_campaign BY $0 USING 'replicated';
-- dump joined_data;

result = FOREACH joined_data GENERATE $0 as campaign, ($3 is null
? 0 : $3) as clicks, $1 as impressions, (double)$3/(double)$1 as
effectiveness:double;
dump result;
```

让我们来看上面的代码。

首先 LOAD 语句指定了数据的位置，以及我们载入数据的格式。通常，Pig 载入的是非规范化的数据。数据的位置是一个 URL。在本地模式执行时，会是一个相对路径。当执行在 MapReduce 模式下，URL 大多数情况下是 HDFS 的一个位置。当 Pig 脚本在 Amazon Web Services（AWS）上执行时，位置信息通常是一个 S3 URL。

在 Load 语句之后的几行中，脚本计算所有去重的点击量。在第一行，它过滤了数据集，只留下包含 True 的数据行，表示这次广告曝光触发了一次点击。在过滤后，每行按

照不同实体去重过滤。然后对所有数据去重后的行按照广告活动进行分组，计算每个广告活动中不同用户的点击行为。结果存储为 of_click_thrus_by_campaign。

这个问题的第二个维度在下面的语句中计算。我们统计每个广告活动的曝光量并不需要 filter。结果存在 count_of_impressions_by_campaign 中。

执行 Pig 脚本会得出下面的输出：

```
(campaign6,2,4,0.5)
(campaign7,4,7,0.5714285714285714)
(campaign8,1,1,1.0)
(campaign10,0,2,)
```

输出中的第一个元素是广告活动的标识符。跟在后面的是按用户去重的点击量和总的曝光量。最后一个元素是效率，是去重点击量占总曝光量的比例。

9.5.2 执行实时分析

现在，让我们将批处理转化为实时分析。一个解释 Pig 脚本的 topology 可能和下面类似：

```
Stream inputStream = topology.newStream("clickthru", spout);
Stream click_thru_stream = inputStream.each(
new Fields("cookie", "campaign", "product", "click"),
new Filter("click", "true"))
.each(new Fields("cookie", "campaign", "product", "click"),
new Distinct())
                .groupBy(new Fields("campaign"))
                .persistentAggregate(
new MemoryMapState.Factory(), new Count(),
new Fields("click_thru_count"))
                .newValuesStream();

Stream impressions_stream = inputStream.groupBy(
new Fields("campaign"))
                .persistentAggregate(
new MemoryMapState.Factory(), new Count(),
new Fields("impression_count"))
                .newValuesStream();

topology.join(click_thru_stream, new Fields("campaign"),
impressions_stream, new Fields("campaign"),
  new Fields("campaign", "click_thru_count", "impression_count"))
                .each(new Fields("campaign",
"click_thru_count", "impression_count"),
new CampaignEffectiveness(), new Fields(""));
```

在前面的 topology 中，我们将数据流分为了两个分支：click_thru_stream 和 impressions_stream。click_thru_stream 中包含的是去重的点击统计量，impressions_stream 中包含了总的曝光量。这两个数据流通过 topology.join 方法做 join 操作。

前面 topology 的问题是 join 操作。在 Pig 中，因为数据集合是静态的，很容易做 join 操作。在 Storm 中做 Join 操作是在每批数据的基础上进行的。这一般不会引起问题。然而，这个 join 操作是一个 inner join，意味着只有两个数据流之间有相关的 tuples 才会向后发射记录。这个例子中，我们过滤了 click_thru_stream 之间的数据，因为我们只想要去重后的点击记录。这样，这个数据流中的计数会比 impressions_stream 中的要少，意味着 join 操作中，可能会丢掉部分 tuples。

提示　在离散集合上的 join 操作定义的很完善，还将它们转换为实时数据流上的定义还不明确的。更详细的信息，查看下面 URL：

- https://cwiki.apache.org/confluence/display/PIG/Pig+on+Storm+Proposal
- https://issues.apache.org/jira/browse/PIG-3453

备选方案是，我们可以使用 Trident 状态结构在不同数据流中共享计数。

改正过的 topology 如图 9-15 所示。

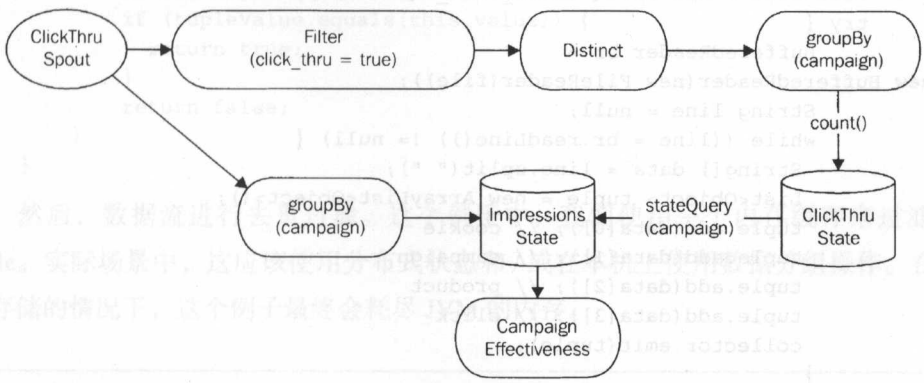


图　9-15

topology 的代码如下所示：

```
StateFactory clickThruMemory = new MemoryMapState.Factory();
ClickThruSpout spout = new ClickThruSpout();
Stream inputStream = topology.newStream("clithru", spout);
```

```
TridentState clickThruState = inputStream.each(
new Fields("cookie", "campaign", "product", "click"),
new Filter("click", "true"))
   .each(new Fields("cookie", "campaign", "product", "click"),
new Distinct())
   .groupBy(new Fields("campaign"))
   .persistentAggregate(clickThruMemory, new Count(),
new Fields("click_thru_count"));

inputStream.groupBy(new Fields("campaign"))
.persistentAggregate(new MemoryMapState.Factory(),
new Count(), new Fields("impression_count"))
.newValuesStream()
.stateQuery(clickThruState, new Fields("campaign"),
new MapGet(), new Fields("click_thru_count"))
.each(new Fields("campaign", "impression_count",
     "click_thru_count"),
new CampaignEffectiveness(), new Fields(""));
```

首先看一下 spout，简单地读取文件，解析每行数据，然后发射 tuple，如下所示：

```
public class ClickThruEmitter
implements Emitter<Long>, Serializable {
...
@Override
public void emitBatch(TransactionAttempt tx,
Long coordinatorMeta, TridentCollector collector) {
     File file = new File("click_thru_data.txt");
     try {
         BufferedReader br =
new BufferedReader(new FileReader(file));
         String line = null;
         while ((line = br.readLine()) != null) {
          String[] data = line.split(" ");
          List<Object> tuple = new ArrayList<Object>();
          tuple.add(data[0]); // cookie
          tuple.add(data[1]); // campaign
          tuple.add(data[2]); // product
          tuple.add(data[3]); // click
          collector.emit(tuple);
         }
         br.close();
     } catch (Exception e) {
         throw new RuntimeException(e);
     }
}
     ...
}
```

在实际的系统中，前面的 spout 一般是从 Kafka 队列中读取数据。或者，spout 可以从 HDFS 中直接读取数据，如果我们试图去代替批处理机制的话。

关于 spout 从 HDFS 中读取数据，已经有一些前期工作可以参考；查看下列 URL 获取更多信息：

https://github.com/jerrylam/storm-hdfs。

为了计算点击的去重计数，topology 首先过滤数据流，只留下触发了点击的曝光记录。Filter 的代码如下：

```
public class Filter extends BaseFilter {
    private static final long serialVersionUID = 1L;
    private String fieldName = null;
    private String value = null;

    public Filter(String fieldName, String value){
        this.fieldName = fieldName;
        this.value = value;
    }

    @Override
    public boolean isKeep(TridentTuple tuple) {
        String tupleValue = tuple.getStringByField(fieldName);
        if (tupleValue.equals(this.value)) {
          return true;
        }
        return false;
    }
}
```

然后，数据流进行去重过滤。这个例子中，我们使用一个内存缓存来过滤重复的 tuple。实际场景中，这应该使用分布式状态和 / 或在本机上使用数据分组操作。在没有持久存储的情况下，这个例子最终会耗尽 JVM 的内存。

对于计算数据流中的近似去重集合的算法，目前已经有一些活跃的尝试。关于 Streaming Quotient Filter（SQF）更多的信息，查看下列 URL：

http://www.vldb.org/pvldb/vol6/p589-dutta.pdf。

我们的例子中，Distinct function 如下面代码所示：

```
public class Distinct extends BaseFilter {
    private static final long serialVersionUID = 1L;
    private Set<String> distincter = Collections.synchronizedSet(new
HashSet<String>());

    @Override
    public boolean isKeep(TridentTuple tuple) {
        String id = this.getId(tuple);
   return distincter.add(id);
    }

    public String getId(TridentTuple t){
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < t.size(); i++){
           sb.append(t.getString(i));
        }
        return sb.toString();
    }
}
```

当获取了所有的去重点击之后，Storm 使用 persistAggregate 方法将信息持久化到 Trident 状态中。这会使用 Count 操作使数据流的规模下降。在这个例子中，我们使用 MemoryMap。但是，一个实际系统中更可能使用一个分布式存储机制比如 Memcache 或者 Cassandra。

处理原始数据流的结果是一个 TridentState 对象，包括了按照广告活动标识符分组的所有的去重点击量 join 两个数据流的关键代码行如下：

```
.stateQuery(clickThruState, new Fields("campaign"),
new MapGet(), new Fields("click_thru_count"))
```

这个操作将原始数据流得出的结果状态和第二个数据流结合在一起。第二个数据流查询状态机制来获取每个活动的去重点击量，并且将结果作为一个字段添加在自身的数据流中，封装为下面的类：

```
public class CampaignEffectiveness extends BaseFunction {
    private static final long serialVersionUID = 1L;

    @Override
    public void execute(TridentTuple tuple, TridentCollector
collector) {
   String campaign = (String) tuple.getValue(0);
        Long impressions_count = (Long) tuple.getValue(1);
        Long click_thru_count = (Long) tuple.getValue(2);
        if (click_thru_count == null)
```

```
            click_thru_count = new Long(0);
        double effectiveness = (double) click_thru_count / (double)
impressions_count;
    Log.error("[" + campaign + "," + String.valueOf(click_thru_count) +
"," + impressions_count + ", " + effectiveness + "]");
    List<Object> values = new ArrayList<Object>();
    values.add(campaign);
    collector.emit(values);
     }
}
```

如上述代码所示，这个类通过计算状态查询结果字段和总量字段的比例来计算广告效果。

9.6　部署 topology

为了部署前面的 topology，我们必须首先使用下列命令获取 Storm-YAML：

```
storm-yarn getStormConfig ../your.yaml --appId
application_1381197763696_0004 --output output.yaml
```

上述命令和特定的 Storm-YARN 示例交互，拉取 storm.yaml 文件，可以用来按照标准机制部署 topology。将 output.yaml 文件拷贝到恰当的位置（一般是 ,~/.storm/storm.yaml）并且按下面命令部署标准的 storm jar 包：

```
storm jar <appJar>
```

9.7　执行 toplogy

执行前面的 topology 会得到下列输出：

```
00:00 ERROR: [campaign10,0,2, 0.0]
00:00 ERROR: [campaign6,2,4, 0.5]
00:00 ERROR: [campaign7,4,7, 0.5714285714285714]
00:00 ERROR: [campaign8,1,1, 1.0]
```

注意这些值和 Pig 脚本执行的结果相同。如果让 topology 运行起来，我们最终会看到效果分值持续下降，如下所示：

```
00:03 ERROR: [campaign10,0,112, 0.0]
00:03 ERROR: [campaign6,2,224, 0.008928571428571428]
00:03 ERROR: [campaign7,4,392, 0.01020408163265306]
00:03 ERROR: [campaign8,1,56, 0.017857142857142856]
```

一个说得通的原因是，我们现在使用了实时系统，持续的消费相同的曝光事件。因为我们只计算去重后的点击量，因为整个中的点击用户在计算中都已经出现过了，所以效果持续会下降。

总结

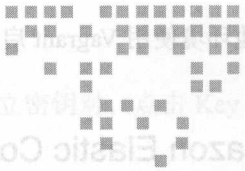
在本章中，我们看到了不同的事情。首先，我们了解了将利用 Pig 进行的批处理机制转换到用 Storm 实现的实时系统的设计方案。我们还了解到，如果将脚本直接进行转换可能会失效，因为实时系统的 join 操作有局限性，传统的 join 操作是基于有限的数据集。我们通过将数据流进行分值并且共享状态来克服这个难题。

其次，可能是最重要的内容，我们介绍了 Storm-YARN；它允许用户重用 Hadoop 的基础设置来部署 Storm。不仅仅使当前的 Hadoop 用户可以快速地过渡到 Storm，而且还允许用户利用 Hadoop 的云机制，比如 Amazon 的 Elastic Map Reduce（EMR）。使用 EMR，Storm 可以快速地部署在云环境下，并且根据需求快速扩容缩容。

最后，作为未来一项工作，社区正在探索能够在 Storm 上直接运行 Pig 的方法。这允许用户直接将现有的分析放在 Storm 上运行。

在这里可以关注到这项工作：https://cwiki.apache.org/confluence/display/PIG/Pig+on+Storm+PropoSal。

在下一章中，我们会探索 Storm 在云环境中使用 Apache Whirr 进行自动部署的方法。虽然没有特别注明，下一章的技术都可以用在云环境的部署中。



第 10 章 Chapter 10

云环境下的 Storm

在本章中，我们会介绍如何在云服务提供商提供的主机环境下部署和运行 Storm。

在第 2 章中，我们介绍了在集群环境下安装 Storm 的必要步骤，并且在后续的章节中，覆盖了补充性技术的安装和配置，比如 Kafka、Hadoop 和 Cassandra。虽然大部分的安装都相对简单，但如果不能彻底适应分布式计算技术，即使维护小规模的集群（就物理设备需求以及配置和维护环境的时间来讲）都会是一个负担。

幸运的是，现在有很多云主机提供商，提供了服务可以按需准备多机器的计算环境。大多数云主机提供商会提供多种服务和选项来满足多数用户的需求，从单节点小空间的服务器到包括了成百上千机器的大规模基础设施。实际上，在因特网内容提供商中一个通用的趋势是选择在云主机提供商那里选择一个内部数据中心。

使用云服务的关键好处是能在必要时按照需求实施或者取消资源部署。例如，一个在线零售商在假期到来时提供额外的服务器和资源以应对需求，当高峰回落时再缩减规模。还有，我们将会看到，云服务提供商提供了一种成本较好的方法对分布式应用进行测试和实现原型。

我们首先使用云服务定制一个 Storm 集群。本章后面，我们会演示如何定制和管理本地、虚拟化的 Storm 实例，可以在完整的集群环境下测试 Storm 应用。

本章包括以下主题：

- 使用 Amazon Web Service（AWS）Elastic Compute Cloud（EC2）来定制虚拟机。
- 使用 Apache Whirr 在 EC2 上自动定制和部署 Storm 集群。

- 在本地环境使用 Vagrant 启动和定制虚拟化 Storm 集群用来进行开发和测试。

10.1 Amazon Elastic Compute Cloud 简介

Amazon 弹性计算云（Elastic Compute Cloud，EC2）是 Amazon 提供的一系列远程计算服务中的核心部分。EC2 允许用户按需租用托管在 Amazon 网络设置上的虚拟计算资源。

我们会从建立 EC2 账户开始，手工启动 Amazon EC2 基础设施上的一个虚拟机。

10.1.1 建立 AWS 帐号

建立一个 AWS 帐号非常容易，但是需要先拥有 Amazon 帐号。如果你还没有 Amazon 帐号，可以在下面建立一个 http://www.amazon.com/。

使用建立的 Amazon 帐号，可以在 http://aws.amazon.com/ 建立一个 AWS 帐号。

10.1.2 AWS 管理终端

AWS 管理终端是 Amazon 提供的所有服务的主要管理接口。我们主要是对 EC2 服务有兴趣，所以我们先登录到 EC2 管理终端，如图 10-1 所示。

图 10-1

建立一个 SSH 密钥对

在启动 EC2 实例之前，首先需要一对密钥。为了建立密钥对，点击 Key Pairs 连接打开密钥对管理界面，如图 10-2 所示。

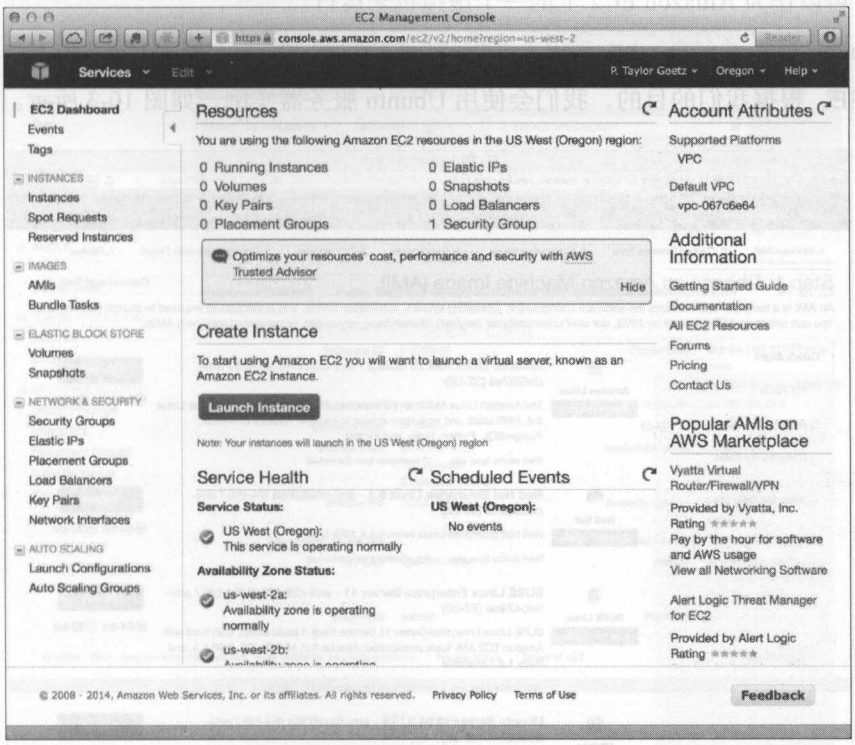


图　10-2

这里会提示你给密钥对一个名字。输入名字后点击 Yes 按钮。这时候，取决于使用的浏览器，会提示下载私钥文件，或者文件会自动下载。

保证私钥文件的安全非常重要，因为对任何使用了这对密钥启动的 EC2 镜像，都能通过私钥文件获得管理员权限。在下载了私钥文件后，需要更改它的文件权限，使其他用户无法读取；例如在 UNIX 环境下，使用下述命令：

```
chmod 400 my-keyfile.pem
```

很多 SSH 客户端会检查密钥文件的访问权限，如果使用的密钥文件是公开可读的，客户端会拒绝使用这个文件或者给出一个告警信息。

10.1.3 手工启动一个 EC2 实例

一旦建好了密钥对，已经准备好启动 EC2 实例了。

启动 EC2 机器的第一步是选择 Amazon Machine Image（AMI）。AMI 是一个虚拟容器的模板，可以作为 Amazon EC2 上的一个虚拟机来运行。

Amazon 提供了多个 AMI 包括了多种常用的操作系统发行版，比如 Red Hat、Ubuntu 以及 SUSE。根据我们的目的，我们会使用 Ubuntu 服务器实例，如图 10-3 所示。

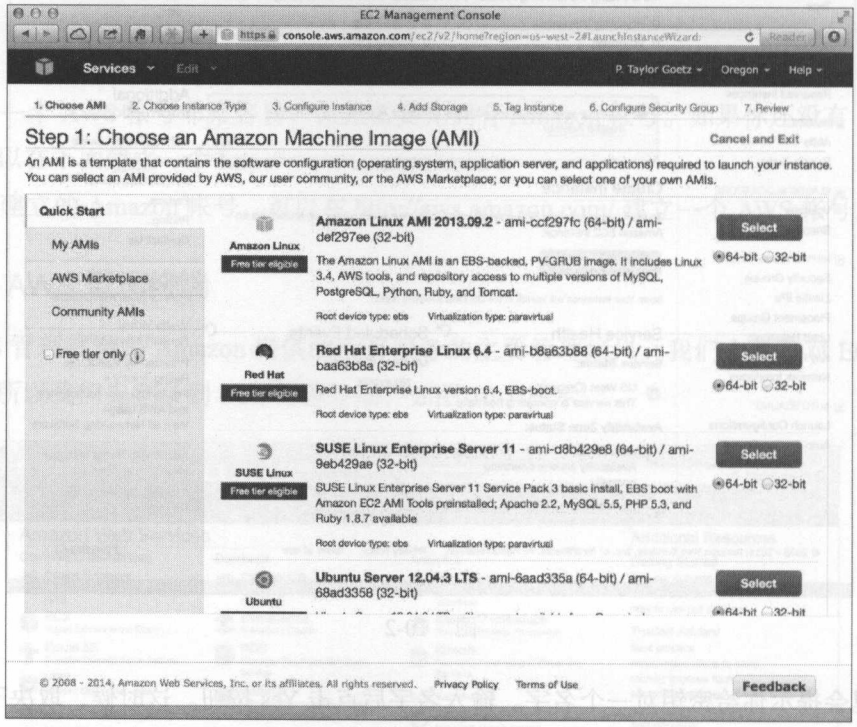


图 10-3

在选择了 AMI 之后，你会看到选择实例类型的提示。实例类型表示了虚拟硬件配置，不同的内存、CPU、存储以及 I/O 性能。Amazon 按照实例运行的小时来收费，从每小时几美分的性能最低实例类型（t1.micro）到每小时几美元的最强大实例类型（hs1.8xlarge）。选择的类型取决于使用场景以及预算。例如，一个 t1.micro 实例（单核 CPU、0.6G 内存，以及低 I/O 性能）可以用作测试目的，但是显然它不能适用于生产环境下的高负载。

在选择了实例类型后，可以通过点击 Review 和 Launch 按钮来启动虚拟机，回顾实例的详细信息，然后点击 Launch 按钮。这里会提示你选择密钥对用来远程登录和控制这个实例。在几分钟后，实例会启动并且如图 10-4 所示运行。

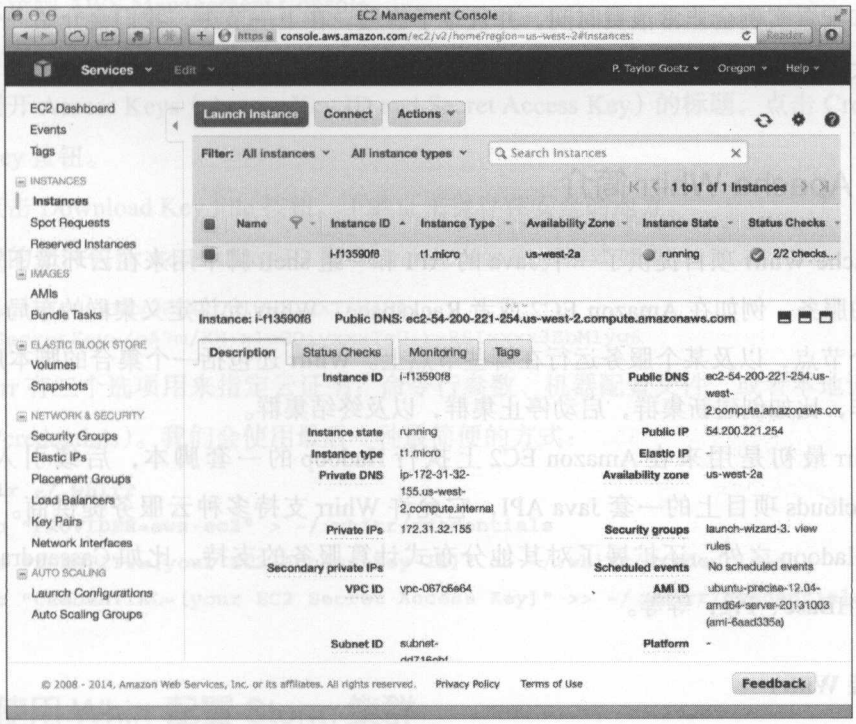


图　10-4

登录 EC2 实例

当启动了一个实例以后，EC2 会在 SHH 安装时，使用你选择的密钥对预先进行设置，允许你远程登录这个虚拟机。为了远程登录虚拟机实例，需要使用到前面下载的私钥文件，以及实例分配的 DNS（或者公网 IP 地址）。你可以在 EC2 管理终端界面点击实例找到并浏览详细信息。

现在可以通过下述命令登录到实例：

```
ssh -i [keypair] [username]@[public DNS or IP]
```

例如，使用 ubuntu 这个用户，使用 my-keypair.pem 密钥文件登录：

```
ssh -i my-keypair.pem ubuntu@ec2-54-200-221-254.us-west-2.compute.
amazonaws.com
```

这个 Ubuntu 用户在远程机器上具有管理员权限，可对该机器随意进行配置。

这时候，可以安装 Storm 或者其他任何需要的服务。然而，在稍微大一点的集群上手工安装和配置实例会非常浪费时间，并且难于管理。在下一节中，我们会介绍一个方法能够在更大的规模下自动进行这个过程。

10.2 Apache Whirr 简介

Apache Whirr 项目提供了一个 Java 的 API 和一组 shell 脚本用来在云环境下安装和启动不同的服务，例如在 Amazon EC2 或者 Rackspace。Whirr 允许定义集群的布局，比如包含多少个节点，以及某个服务运行在哪些节点上。Whirr 还包括一个集合的脚本用来执行管理操作，比如创建新集群，启动停止集群，以及终结集群。

Whirr 最初是用来在 Amazon EC2 上执行 Hadoop 的一套脚本，后续引入了基于 Apache jclouds 项目上的一套 Java API，它允许 Whirr 支持多种云服务提供商。Whirr 除了支持 Hadoop 之外，还扩展了对其他分布式计算服务的支持，比如 Cassandra、Elastic Search、HBase、Pig，等等。

安装 Whirr

首先下载最新的发行版，并且将它解压缩到你用来创建和管理集群的机器上：

```
wget http://www.apache.org/dist/whirr/whirr-0.8.2/whirr-0.8.2.tar.gz
tar -zxf whirr-0.8.2.tar.gz
```

为了方便起见，我们添加 Whirr 的 bin 目录到系统的 PATH 环境变量中，这样就可以直接像下面这样运行 Whirr 命令了：

```
WHIRR_HOME=/Users/tgoetz/whirr-0.8.2
export PATH=$PATH:$WHIRR_HOME/bin
```

Whirr 使用 SSH 和云实例通信，所以我们需要建立一个专用的密钥对供 Whirr 使用。Whirr 需要密钥使用空密码，如下面命令所示：

```
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa_whirr
```

为了让 Whirr 使用云服务帐号进行交互，它需要知道你的证书。对 EC2 来讲，这个包

括 EC2Access Key ID 和 EC2 Secret Access Key。如果你的 AWS 帐号是新建的，需要生成新的证书；否则，你可以直接将证书下载到安全的地方。为了生成一套新的 EC2 证书，执行下列步骤：

1. 登录到 AWS Management Console。

2. 击右上角导航栏，选中 Security Credentials。

3. 展开 Access Keys（Access Key ID and Secret Access Key）的标题，点击 Create New Access Key 按钮。

4. 点击 Download Key File 按钮，下载证书保存在安全的地方。

下载的文件包括 Access Key ID 和 Secret Access Key 格式如下：

```
AWSAccessKeyId=QRIXIUUTWRXXXXTPW4UA
AWSSecretKey=/oA7m/XW+x1eGQiyxxxTsU+rxRSIxxxx3EbM1yg6
```

Whirr 有三个选项用来指定云证书：命令行参数、机器配置文件，或者本地证书文件（~/.whirr/credentials）。我们会使用最后一种最简便的方式：

```
mkdir ~/.whirr
echo "PROVIDER=aws-ec2" > ~/.whirr/credentials
echo "IDENTITY=[your EC2 Access Key ID]" >> ~/.whirr/credentials
echo "CREDENTIAL=[your EC2 Secret Access Key]" >> ~/.whirr/credentials
```

10.3　使用 Whirr 配置 Storm 集群

现在已经安装了 Whirr，让我们来看一下集群配置。Whirr 的配置文件或方法，就是使用 Java 的属性文件来包含 Whirr 的属性，定义了节点的布局已经集群内的服务。

让我们首先看一下启动一个 3 个节点 ZooKeeper 集群所必须的最小配置：

```
whirr.cluster-name=zookeeper
whirr.instance-templates=3 zookeeper
```

whirr.cluster-name 属性给该 ZooKeeper 集群指定唯一标识符，在运行管理命令时使用，比如列出一个集群中的节点，或者毁掉一个集群。

whirr.instance-template 属性定义了集群中节点个数，以及每个节点上运行的服务。前面的例子中，我们定一个三节点集群，每个节点指定作为一个 ZooKeeper 的角色。

定义了这些属性之后，我们已经足以告诉 Whirr 如何启动并且管理一个 ZooKeeper 集

群。Whirr 给其他选项会使用默认值。然而，还有一些选项一般是需要覆盖的。例如，我们想让 Whirr 使用我们前面创建的专用密钥对，如下面代码段所示：

```
whirr.private-key-file=${sys:user.home}/.ssh/id_rsa_whirr
whirr.public-key-file=${whirr.private-key-file}.pub
```

还有，我们还会配置 Whirr 适应特定的硬件配置，以及我们的集群放在什么域下，如下面代码段所示：

```
whirr.image-id=us-east-1/ami-55dc0b3c
whirr.hardware-id=t1.micro
whirr.location=us-east-1
```

whirr.image-id 属性根据不同的云服务提供商是不一样的，指定了我们私用的哪种镜像。这里，我们已经指定了使用 Ubuntu 10.04 64-bit AMI。因为我们只是测试 Whirr，所以选择了最小化（最便宜）的实例类型。最后，我们指定集群需要部署在 us-east-1 域下。

为了获取 AMI 公开的完整列表，执行下面的步骤：

1. 从 Ec2 管理终端，在右上角选择下拉菜单的趋于。

2. 在左边的导航面板，点击 AMI。

3. 在页面上上方的 Filter 下拉选项中，选择 Public images。

Whirr 在 Ubuntu Linux 镜像上已经有比较充分的测试。其他操作系统可能也可以正常工作，如果你遇到了一些问题，尝试一下 Ubuntu 镜像。

启动集群

我们给 ZooKeeper 集群的配置如下面的代码段所示：

```
whirr.cluster-name=zookeeper
whirr.instance-templates=3 zookeeper
whirr.private-key-file=${sys:user.home}/.ssh/id_rsa_whirr
whirr.public-key-file=${whirr.private-key-file}.pub
whirr.image-id=us-east-1/ami-55dc0b3c
whirr.hardware-id=t1.micro
whirr.location=us-east-1
```

如我们将这些属性存储到 zookeeper.properties 文件中，我们可以通过下面的命令来启动 ZooKeeper 集群：

```
whirr launch-cluster --config zookeeper.properties
```

当命令执行时，Whirr 会输出新建的 ZooKeeper 实例的列表，以及我们可以用来连接

这些实例的 SSH 命令。

你可以通过下面这些 SSH 命令连接这些实例：

```
[zookeeper]: ssh -i /Users/tgoetz/.ssh/id_rsa_whirr -o
"UserKnownHostsFile /dev/null" -o StrictHostKeyChecking=no
storm@54.208.197.231
[zookeeper]: ssh -i /Users/tgoetz/.ssh/id_rsa_whirr -o
"UserKnownHostsFile /dev/null" -o StrictHostKeyChecking=no
storm@54.209.143.46
[zookeeper]: ssh -i /Users/tgoetz/.ssh/id_rsa_whirr -o
"UserKnownHostsFile /dev/null" -o StrictHostKeyChecking=no
storm@54.209.22.63
```

要销毁一个集群，使用和建立集群同样的配置运行 whirr destroy-cluster 命令。

当你用完集群，可以通过下列命令终止所有的实例：

```
whirr destroy-cluster --config zookeeper.properties
```

10.4　Whirr Storm 简介

Whirr Storm 项目（https://github.com/ptgoetz/whirr-storm）是一个用来配置 Storm 集群的 Whirr 服务的实现。Whirr Storm 支持基于 Storm 配置文件 storm.yaml 配置所有的 Storm 守护进程以及进行控制操作。

安装 Whirr Storm

安装 Whirr Storm 服务很简单，只需要将 Jar 包放到 $WHIRR_HOME/lib 目录下：

```
wget http://repo1.maven.org/maven2/com/github/ptgoetz/whirr-
storm/1.0.0/whirr-storm-1.0.0.jar -P $WHIRR_HOME/lib
```

下一步，运行不带参数 Whirr 命令检查安装是否成功，会打印出 Whirr 中可以使用的角色列表。这个列表现在应该包括 Whirr Storm 提供的角色，如下所示：

```
$ whirr
…
  storm-drpc
  storm-logviewer
  storm-nimbus
  storm-supervisor
  storm-ui
```

集群配置

在我们前面的 Whirr 例子中，我们建立了一个三节点的集群，每个节点包括一个 ZooKeeper 的角色。Whirr 还允许将多个角色指派给一个节点，这个特性我们在建立 Storm 集群中是需要的。在我们深入到使用 Whirr 配置 Storm 之前，先来看一下 Whirr Storm 中定义的不同的角色，如表 10-1 所示。

表 10-1

角　色	说　明
storm-nimbus	用来运行 Nimbus 守护进程，每个集群中只有一个节点可以指定为这个角色
storm-supervisor	用来运行 Supervisor 守护进程
storm-ui	用来运行 Storm UI 的 Web 服务
storm-logviewer	用来运行 Storm 的 logviewer 服务。这个角色能且只能和 storm-supervisor 角色指定在同节点上
storm-drpc	用来运行 Storm DRPC 服务
zookeeper	是由 Whirr 提供的，这个角色的节点是一个 ZooKeeper 集群的一部分。在 Storm 集群中必须有一个 ZooKeeper 节点，多节点 ZooKeeper 集群必须有奇数个数的节点

要在 Whirr 配置中使用这些角色，我们需要在 whirr.instance-template 属性中按照如下格式指定角色：

```
whirr.instance-templates=[# of nodes] [role 1]+[role 2],[# of nodes]
[role 3]+[role n]
```

例如，我们需要建立一个单节点伪集群，所有的 Storm 守护进程都要运行在一个机器上，需要给 whirr.instance-template 使用下面的值：

```
whirr.instance-template=1 storm-nimbus+storm-ui+storm-logviewer+storm-
supervisor+zookeeper
```

如果我们想建立一个多节点集群，一个节点运行 Nimbus 和 Storm UI，三个节点运行 supervisor 和 logviewer 守护进程，还有一个 3 节点的 ZooKeeper 集群，应使用下列配置：

```
whirr.instance-templates=1 storm-nimbus+storm-ui,3 storm-
supervisor+storm-logviewer, 3 zookeeper
```

自定义 Storm 的配置

Whirr Storm 会生成一个 storm.yaml 配置文件，其中包括 nimbus.host，storm.zookeeper.servers，以及 drpc.serversd 的值，它们由集群中节点被赋值的角色计算而来。所有的 Storm 其他的配置参数会阶乘默认值，除非是特别被指定覆盖。注意，如果你尝试覆盖

nimbus.host、storm.zookeeper.servers 或者 drpc.servers，Whirr Storm 会忽略这种覆盖并且生成一条告警日志消息。

提示　虽然 Whirr Storm 会为集群自动生成和配置 nimbus.host 值，在本地执行命令时，你仍然需要告诉 Storm 可执行程序 Nimbus 所在的主机名。当你有一个多节点集群时，最简单和最方便的方法是通过使用 -c 命令为 nimbus 指定主机名：

```
Storm <command> [arguments] -c nimbus.host=<nimbus
hostname>
```

其他 Storm 配置参数可以在 Whirr 配置文件中添加前缀为 whirr-storm 的属性来指定。例如，为了指定 topology.message.timeout.secs 参数，需要在 Whirr 配置文件中添加下面的值：

```
whirr-storm.topology.message.timeout.secs=30
```

前面的代码会在 storm.yaml 文件中生成下列结果：

```
topology.message.timeout.secs: 30
```

在 Whirr 配置文件中添加可以接收一个列表值的参数，参数值使用逗号隔开，例如 supervisor.slots.ports 的端口配置：

```
whirr-storm.supervisor.slots.ports=6700,6701,6702,6703
```

前面的代码会生成下面的 YAML 选项：

```
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
```

自定义防火墙策略

EC2 上创建一个机器实例，大部分的网络端口都默认由防火墙阻断。为了允许集群中实例间的通信，你需要明确配置防火墙规则，允许特定机器和端口的进出流量。

默认情况下，Whirr Storm 会自动建立 Storm 组件用来通信必须的安全组和防火墙规则，例如开启 Nimbus 的 Thrift 端口用来接收 topology 提交，在 Nimbus 和 Supervisor 节点到 ZooKeeper 节点之间的 2181 端口，如图 10-5 所示。

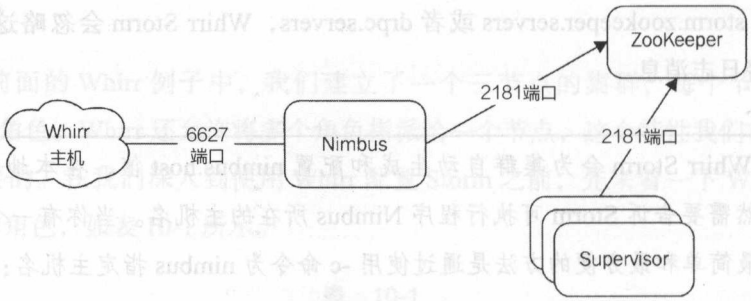


图 10-5

然而，很多情况下，Storm worker 进程需要在一个随机分配的端口上和其他服务进行通信。例如，如果你有一个 spout 从外部队列消费数据，或者 bolt 写入一个数据库，则需要额外的防火墙规则放开这种交互。

考虑这样一个场景，我们有一个 spout 从 Kafka 队列中读取数据并且发送数据流到 bolt 中，这个 bolt 会将数据写入 Cassandra 数据库。这个场景中，我们需要如下配置 whirr.instance-template 值：

```
whirr.instance-templates=3 kafka,3 cassandra,1 storm-nimbus,3 storm-
supervisor, 3 zookeeper
```

安装完成后，我们需要修改防火墙配置允许每个 Supervisor/worker 节点都能和 Kafka 节点的 9092 端口，以及 Cassandra 节点的 9126 端口通信，如图 10-6 所示。

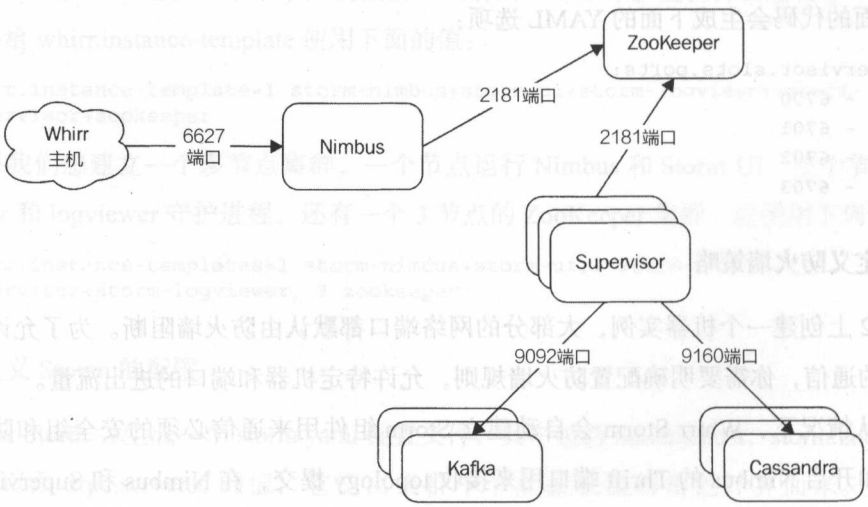


图 10-6

这种情况下，Whirr Storm 的配置属性 whirr.storm.supervisor.firewall-rules 允许你在集群的其他节点开启任意的端口。属性值是用好分隔开的端口 - 端口对，如下所示：

```
whirr.storm.supervisor.firewall-rules=[role1]:[port1],[role2]:[port2]
```

例如，为这个场景添加的规则，我们需要使用下面的设置：

```
whirr.storm.supervisor.firewall-rules=cassandra:9160,kafka:9092
```

这个配置会指示 Whirr Storm 建立防火墙规则允许每个 Supervisor 节点连接到每个 Cassandra 的 9160 端口，每个 Supervisor 节点可以连到 Kafka 节点的 9092 端口。

10.5　Vagrant 简介

Vagrant（http://www.vagrantup.com/）是一个和 Apache Whirr 类似的工具，用于辅助采用简单和复用的方法来准备虚拟机实例。然而，Whirr 和 Vagrant 有一点关键的不同。Whirr 的主要目的是在云环境下准备虚拟机，Vagrant 更多是作用在本地虚拟化上，配合虚拟化软件（比如 VirtualBox 和 VMWare）。

Vagrant 支持多种虚拟机载体，包括 VirtualBox（https://www.virtualbox.org）和 VMWare（http://www.vmware.com）。在本章中，我们会介绍 Vagrant 和 VirtualBox 结合的使用方法，因为 VirtualBox 是免费的，并且很好地被 Vagrant 支持。

在使用 Vagrant 之前，必须首先安装 4.x 版本的 VirtualBox（Vagrant 不支持 5.x 版本）。我们在第 2 章中已经介绍了 VirtualBox 的安装，这里就不再重复。安装 VirtualBox 基本上就是运行一个安装程序，如果你遇到问题，请参考第 2 章。

10.5.1　安装 Vagrant

Linux 的安装包和 OS X 和 Windows 下的 Vagrant 安装程序在 Vagrant 官网可以获取（http://www.vagrantup.com/downloads.html）。确保安装最新的 Vagrant 版本，因为最新版包括最近的更新和 bug 修复。安装进程会更新系统的 PATH 变量，将 Vagrant 可执行程序包括进来。你可以通过在命令行下执行 vagrant –version 来确认安装是否成功：

```
$ vagrant --version
Vagrant 1.3.5
```

如果命令因为某种原因执行失败，去 Vagrant 官网查阅常见问题的解决方案。

10.5.2 创建第一个虚拟机

使用 Vagrant 创建一个虚拟机包括两个步骤。首先，使用 vagrant init 命令初始化一个 Vagrant 项目：

```
$ vagrant init precise64 http://files.vagrantup.com/precise64.box
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

vagrant init 的两个命令是 Vagrant box 的 name 和 URL。Vagrant box 是一个使用 Vagrant 打包的虚拟机镜像。因为 Vagrant box 可能很大（超过 300MB），Vagrant 会将其存在本地硬盘，而不是每次都下载。name 参数给 box 提供了一个简单的标识符，它可以在其他 Vagrant 配置重用，URL 参数告诉 Vagrant 从哪里下载 box。

下一步是创建一个虚拟机：

```
$ vagrant up
```

如果 vagrant init 命令指定的 Vagrant box 在本地硬盘上没找到，Vagrant 会去下载它。Vagrant 然后会克隆该虚拟机，启动并配置网络使它可以从本地机器访问。当命令执行完毕后，一个运行着 Ubuntu 12.04 LTS 64-bit 的 VirtualBox 虚拟机在就在后台运行了。

你可以通过 SSH 命令登录这个机器：

```
$ vagrant ssh
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:23:18 2012 from 10.0.2.2
vagrant@precise64:~$
```

Vagrant 有管理员权限，所以你可以在虚拟机上做任何事情，比如安装软件包或者修改文件。当你用完了该虚拟机之后，可以通过执行 vagrant destroy 关闭并且删除该虚拟机的痕迹：

```
$ vagrant destroy
Are you sure you want to destroy the 'default' VM? [y/N] y
[default] Forcing shutdown of VM...
[default] Destroying VM and associated drives...
```

Vagrant 提供了额外的管理命令，用来执行例如挂起、恢复、停止虚拟机等操作。执行 vagrant –help 命令，可以看到 Vagrant 提供命令概览。

Vagrantfile 和共享的文件系统

当运行 vagrant init 命令时，Vagrant 在命令执行目录下建立了一个叫做 Vagrantfile 的文件。这个文件描述了一个项目需要的机器类型以及如何安装机器。Vagrantfiles 是用 Ruby 语法写的，即使你不是 Ruby 开发者也很容易看懂。Vagrantfile 的初始内容非常少，基本都由文档注释组成。移掉注释符，我们的 Vagrant 文件看起来如下所示：

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "precise64"
  config.vm.box_url = "http://files.vagrantup.com/precise64.box"
end
```

你可以看到，这个文件简单地包括了我们传递给 vagrant init 命令的 box 名字和 URL。我们在后面使用 Vagrant 项目准备一个虚拟化 Storm 集群时，会扩展这个文件。

当你使用 vagrant up 创建一个虚拟机时，Vagrant 默认会在虚拟机上创建一个共享文件夹（/vagrant），和项目目录（包含了 Vagrantfile 的目录）中的内容同步。你可以测试这个功能，登录虚拟机然后在该目录下列出内容：

```
$ vagrant ssh
vagrant@precise64:~$ ls /vagrant/
Vagrantfile
```

这里是我们存储所有准备脚本和数据文件的地方。vagrant destroy 命令会删除一个虚拟机所有记录，它不会删除该项目目录下的内容。这允许我们把持久化的项目数据存储起来，对我们所有的虚拟机总是可用。

Vagrant 安装准备

Vagrant 支持通过脚本或者 Puppet 和 Chef 进行安装准备。我们会使用脚本进行准备，因为这是最简单的方法，除了了解基础的 shell 脚本之外不需要任何额外的知识。

为了实例使用 Vagrant shell 如何进行准备工作，我们会修改 Vagrant 项目在 Vagrant 虚拟机上安装一个 Apache Web 服务器。首先建立一个简单的脚本，使用 Ubuntu 的 APT 包管理命令来安装一个 Apache2。将命令保存到 install_apache.sh 脚本中，和 Vagrantfile 处在

同一个目录：

```
#!/bin/bash
apt-get update
apt-get install -y apache2
```

下一步，修改 Vagrantfile 增加下列行，让 Vagrant 在安装准备时执行我们的脚本：

```
config.vm.provision "shell", path: "install_apache.sh"
```

最后，配置端口转发，让母机上的 80 端口的请求可以转发到虚拟主机上的 8080 端口：

```
config.vm.network "forwarded_port", guest: 80, host: 8080
```

修改完成的 Vagrantfile 现在像下面这样：

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "precise64"
  config.vm.box_url = "http://files.vagrantup.com/precise64.box"
  config.vm.provision "shell", path: "install_apache.sh"
  config.vm.network "forwarded_port", guest: 80, host: 8080
end
```

如果虚拟机仍然在运行，通过执行 vagrant destroy 杀死进程，然后执行 vagrant up 命令建立一个新的虚拟机。Vagrant 执行完毕后，你可以在母机上执行 http://localhost:8080 访问到 Apache 的默认页面。

使用 Vagrant 配置虚拟机集群

为了使用 Vagrant 模板化安装一个虚拟的 Storm 集群，需要一种方式在一个 Vagrant 项目中配置多个虚拟机。幸运的是，Vagrant 语法支持多个虚拟机，这使得很容易将单机器的项目转化为一个多机器的配置。

为了多机器安装，我们会定义两个虚拟机，名字是 www1 和 www2。为了避免在母机上的端口冲突，我们将主机上的 8080 转发到 www1 上的 80 端口，将母机上的 7070 转发到 www2 上的 80 端口，配置代码如下所示：

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "www1" do |www1|
    www1.vm.box = "precise64"
```

```
    www1.vm.box_url = "http://files.vagrantup.com/precise64.box"
    www1.vm.provision "shell", path: "apache.sh"
    www1.vm.network "forwarded_port", guest: 80, host: 8080
  end
  config.vm.define "www2" do |www2|
    www2.vm.box = "precise64"
    www2.vm.box_url = "http://files.vagrantup.com/precise64.box"
    www2.vm.provision "shell", path: "apache.sh"
    www2.vm.network "forwarded_port", guest: 80, host: 7070
  end
end
```

多虚拟机的安装，不带参数运行 vagrant up 命令会启动定义在 Vagrantfile 中的每个机器。这个特性对 Vagrant 的其他命令也都适用。为了控制某个单独的虚拟机，添加虚拟机的名称到命令行中。例如，如果我们只想创建 www1 虚拟机，可使用下面的命令：

```
vagrant up www1
```

同样，如果我们想销毁某个虚拟机，可以执行下列命令：

```
vagrant destroy www1
```

10.6　生成 Storm 安装准备脚本

在第 2 章中，我们讲了如何在 Ubuntu Linux 上手工安装 Storm 以及它的依赖工具。我们利用第 2 章中的安装命令，使用它们建立一个 Vagrant 安装准备脚本，将手工步骤自动化。如果你不明白安装准备脚本中的命令，参考第 2 章中的内容查看更详细的解释。

10.6.1　ZooKeeper

ZooKeeper 在大部分 Linux 平台下都有预打包好的安装包，这简化了我们的安装脚本，让安装包管理程序来做大部分工作。下面的命令安装了 ZooKeeper：

```
install-zookeeper.sh
```

添加如下命令来安装 ZooKeeper：

```
apt-get update
apt-get --yes install zookeeper=3.3.5* zookeeperd=3.3.5*
```

10.6.2 Storm

Storm 安装脚本稍微复杂点，因为打包的程序必须手工安装。我们会采用第 2 章中用过的命令，将它们汇总到脚本中，参数化这些命令，将 Storm 的版本号作为脚本的参数。这允许我们在不同版本的 Storm 中切换，而不用修改安装脚本。代码片段如下所示：

install-storm.sh

```
apt-get update
apt-get install -y unzip supervisor openjdk-6-jdk

/etc/init.d/supervisor stop

groupadd storm
useradd --gid storm --home-dir /home/storm --create-home --shell /bin/
bash storm
unzip -o /vagrant/$1.zip -d /usr/share/
chown -R storm:storm /usr/share/$1
ln -s /usr/share/$1 /usr/share/storm
ln -s /usr/share/storm/bin/storm /usr/bin/storm

mkdir /etc/storm
chown storm:storm /etc/storm

rm /usr/share/storm/conf/storm.yaml
cp /vagrant/storm.yaml /usr/share/storm/conf/
cp /vagrant/cluster.xml /usr/share/storm/logback/
ln -s /usr/share/storm/conf/storm.yaml /etc/storm/storm.yaml

mkdir /var/log/storm
chown storm:storm /var/log/storm
```

install-storm.sh 脚本利用了 Vagrant 共享目录（/vagrant），这很方便地将 storm.yaml 和 logback.xml 配置文件和 Vagrantfile 放在一起。

在 storm.yaml 文件中，我们会使用主机名代替 IP 地址，可以用 Vagrant 来配置主机名解析，如下面代码片段所示：

storm.yaml

```
storm.zookeeper.servers:
    - "zookeeper"

nimbus.host: "nimbus"

# netty transport
storm.messaging.transport: "backtype.storm.messaging.netty.Context"
```

```
storm.messaging.netty.buffer_size: 16384
storm.messaging.netty.max_retries: 10
storm.messaging.netty.min_wait_ms: 1000
storm.messaging.netty.max_wait_ms: 5000

drpc.servers:
  - "nimbus"
```

10.6.3 Supervisord

install-storm.sh 脚本会安装 supervisord 程序，但我们仍需要配置它来管理 Storm 的守护进程。我们不会给每个守护进程都单独新建一个配置文件，而是写一个脚本来生成 supervisord 配置项的文件，服务作为参数名，如下面代码段所示：

configure-supervisord.sh

```
echo [program:storm-$1] | sudo tee -a /etc/supervisor/conf.d/storm-$1.
conf
echo command=storm $1 | sudo tee -a /etc/supervisor/conf.d/storm-$1.
conf
echo directory=/home/storm | sudo tee -a /etc/supervisor/conf.d/
storm-$1.conf
echo autorestart=true | sudo tee -a /etc/supervisor/conf.d/storm-$1.
conf
echo user=storm | sudo tee -a /etc/supervisor/conf.d/storm-$1.conf
```

configure-supervisord.sh 脚本接收一个参数，参数是 Storm 服务的名称。例如，要给 Nimbus 守护进程新建一个 supervisord 的配置文件，可以执行下面的命令：

```
sh configure-supervisord.sh nimbus
```

Storm Vagrant

在 Storm 集群中，我们会建立一个单节点 ZooKeeper、一个 Nimbus 节点、一个或多个 Supervisor 节点。因为 Vagrantfile 使用 Ruby 语言编写，我们已经接触过 Ruby 语言的很多特性，这些特性允许我们指定更健壮的配置文件。例如，我们可以随意调整 Supervisor 节点的个数。

在 storm.yaml 文件中，我们使用主机名而不是 IP 地址，意味着我们的机器必须能够将主机名解析到 IP 地址。Vagrant 本身并没有管理 /etc/hosts 文件的能力，但幸运的是，有一个 Vagrant 插件可以做到。在我们研究 Storm 集群的 Vagrantfile 之前，首先安装一下 vagrant-hostmanager 插件（https://github.com/smdahlen/vagrant-hostmanager），使用下列命令：

```
vagrant plugin install vagrant-hostmanager
```

vagrant-hostmanager 插件会在集群的所有机器上配置主机名解析。还有个可选项在母机和虚拟机之间添加主机名解析。

下一步，我们来一行一行地看一下完整的 Vagrantfile 文件：

```
require 'uri'
# Configuration
STORM_DIST_URL = "https://dl.dropboxusercontent.com/s/dj86w8ojecgsam7/
storm-0.9.0.1.zip"
STORM_SUPERVISOR_COUNT = 2
STORM_BOX_TYPE = "precise64"
# end Configuration

STORM_ARCHIVE = File.basename(URI.parse(STORM_DIST_URL).path)
STORM_VERSION = File.basename(STORM_ARCHIVE, '.*')

# Vagrantfile API/syntax version. Don't touch unless you know what
you're doing!
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.hostmanager.manage_host = true
  config.hostmanager.enabled = true
  config.vm.box = STORM_BOX_TYPE

  if(!File.exist?(STORM_ARCHIVE))
    `wget -N #{STORM_DIST_URL}`
  end

  config.vm.define "zookeeper" do |zookeeper|
    zookeeper.vm.network "private_network", ip: "192.168.50.3"
    zookeeper.vm.hostname = "zookeeper"
    zookeeper.vm.provision "shell", path: "install-zookeeper.sh"
  end

  config.vm.define "nimbus" do |nimbus|
    nimbus.vm.network "private_network", ip: "192.168.50.4"
    nimbus.vm.hostname = "nimbus"
    nimbus.vm.provision "shell", path: "install-storm.sh", args:
STORM_VERSION
    nimbus.vm.provision "shell", path: "config-supervisord.sh", args:
"nimbus"
    nimbus.vm.provision "shell", path: "config-supervisord.sh", args:
"ui"
    nimbus.vm.provision "shell", path: "config-supervisord.sh", args:
"drpc"
    nimbus.vm.provision "shell", path: "start-supervisord.sh"
```

```
  end

  (1..STORM_SUPERVISOR_COUNT).each do |n|
    config.vm.define "supervisor#{n}" do |supervisor|
      supervisor.vm.network "private_network", ip: "192.168.50.#{4 +
n}"
      supervisor.vm.hostname = "supervisor#{n}"
      supervisor.vm.provision "shell", path: "install-storm.sh", args:
STORM_VERSION
      supervisor.vm.provision "shell", path: "config-supervisord.sh",
args: "supervisor"
      supervisor.vm.provision "shell", path: "config-supervisord.sh",
args: "logviewer"
      supervisor.vm.provision "shell", path: "start-supervisord.sh"
    end
  end
end
```

文件的第一行告诉 Ruby 解析需要的 uri 模块，我们会用来做 URL 解析。

下一步，建立一些变量表示 Storm 发行版获取的 URL，我们需要的 Supervisor 节点的个数，以及我们虚拟机的 Vagrant box 类型。这些变量都可以由用户更改。

STORM_ARCHIVE 和 STORM_VERSION 变量用来保存 Storm 发行版本的文件名和版本名，值是使用 Ruby 的 File 和 URI 类从发行版的 URL 中解析出来的，这些值会作为参数传递给安装准备脚本。

下一步是 Vagrant 配置的主要部分。首先配置 vagrant-hostmanager 插件：

```
config.hostmanager.manage_host = true
config.hostmanager.enabled = true
```

这里，告诉 vagrant-hostmanager 插件管理母机和虚拟机之间的主机名解析，它还应该管理虚拟机的 /etc/hosts 文件。

下一段会检查 Storm 发行版的压缩包是否已经下载完成；如果没有，它使用 wget 命令来下载压缩包，如下面代码片段所示：

```
if(!File.exist?(STORM_ARCHIVE))
  `wget -N #{STORM_DIST_URL}`
end
```

前面的代码会下载 Storm 的压缩包到 Vagrantfile 文件所在的目录，准备安装的脚本可以在 /vagrant 共享目录里访问到压缩包。

下两段代码配置了 ZooKeeper 和 Nimbus，意思非常直白，包括了两个我们之前没见过的新的指令：

```
storm jar myTopology.jar com.example.MyTopology my-topology -c nimbus.
host=nimbus
```

zookeeper.vm.network 指令指示 Vagrant 分配一个特定的 IP 地址给使用了 VirtualBox host-only 网络转发的虚拟机。下一行告诉 Vagrant 给特定虚拟机的主机名赋值。最后，给每个节点引入适当的安装脚本。

最后一段配置了 Supervisor 节点。Ruby 代码创建一个循环，从 1 遍历到 STORM_SUPERVISOR_COUNT，这个值是 Supervisor 的个数且可以修改。它会动态设置虚拟机的名称、主机名以及 IP 地址，基于 STORM_SUPERVISOR_COUNT 设定的 Supervisor 节点的个数。

创建 Storm 集群

到现在已经定义好了集群的 Vagrantfile 配置文件，并安装了所需要的脚本，我们现在可以通过 vagrant up 创建 Vagrant 集群了。因为要建立四个虚拟机，以及每个机器都要装一些软件，安装过程会持续一段时间。

一旦 Vagrant 已经完成集群的创建，可以在母机上访问 http://nimbus:8080 查看 Storm 的 UI 界面。要向集群提交一个 topology，可以执行下述命令：

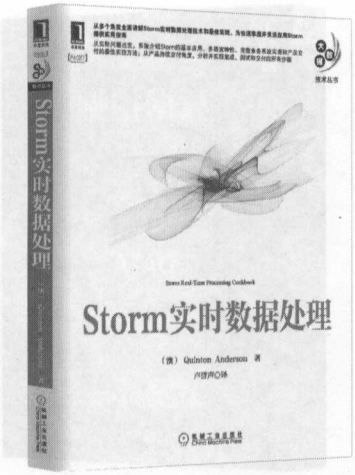
```
storm jar myTopology.jar com.example.MyTopology my-topology -c nimbus.
host=nimbus
```

总结

在本章中，浅显地介绍了如何在云环境下部署 Storm，但是希望能介绍给你更多有效的可行办法。先介绍了在云环境（比如 Amazon EC2）上部署集群，然后介绍了在本地工作站甚至是内部虚拟机提供的云环境上部署集群。

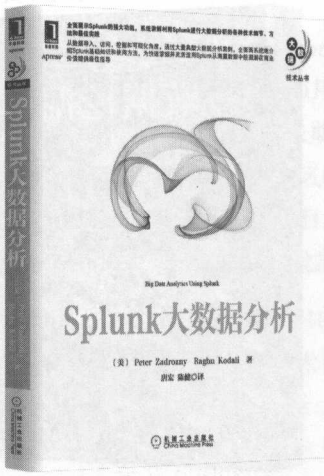
建议继续深入研究云主机提供的服务（比如 AWS）以及虚拟化设置程序（比如 Vagrant），这样能够更好地准备 Storm 安装选项。将第 2 章介绍的手工安装过程和本章介绍的自动安装技术相结合，你能够根据需求找到更好的开发、测试和部署方案。

推荐阅读



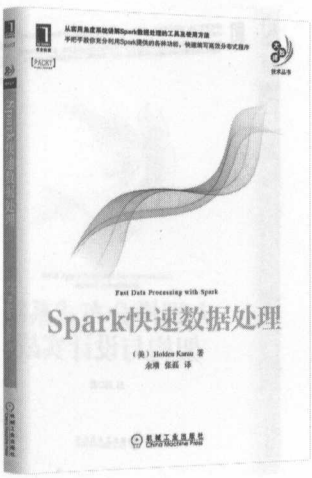
Storm实时数据处理

作者：Quinton Anderson ISBN：978-7-111-46663-5 定价：49.00元



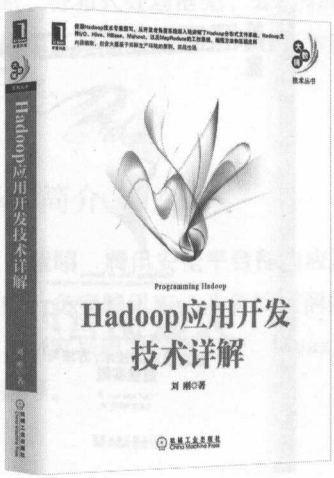
Splunk大数据分析

作者：Peter Zadrozny 等 ISBN：978-7-111-46429-7 定价：69.00元



Spark快速数据处理

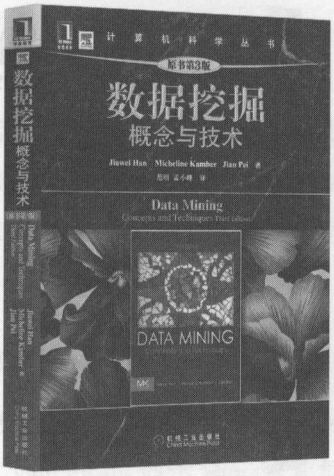
作者：Holden Karau ISBN：978-7-111-46311-5 定价：29.00元



Hadoop应用开发技术详解

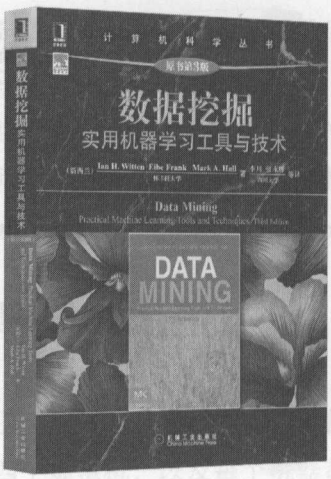
作者：刘刚 ISBN：978-7-111-45244-7 定价：79.00元

推荐阅读



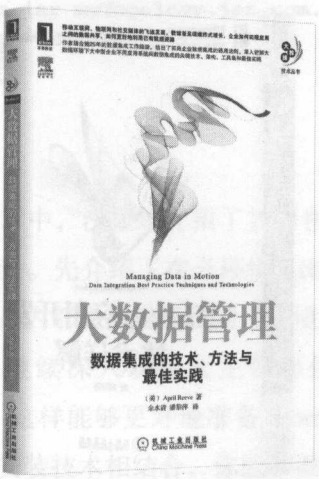
数据挖掘：概念与技术（原书第3版）

作者：Jiawei Han 等 ISBN：978-7-111-39140-1 定价：79.00元



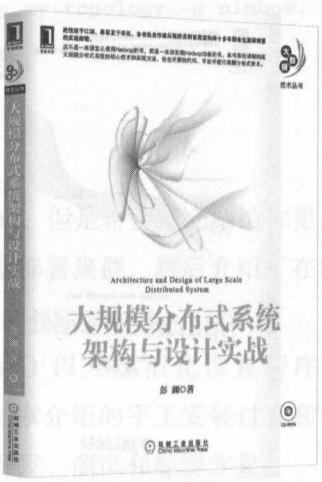
数据挖掘：实用机器学习工具与技术（原书第3版）

作者：Ian H. Witten 等 ISBN：978-7-111-45381-9 定价：79.00元



大数据管理：数据集成的技术、方法与最佳实践

作者：April Reeve ISBN：978-7-111-45905-7 定价：59.00元



大规模分布式系统架构与设计实战

作者：彭渊 ISBN：978-7-111-45503-5 定价：59.00元

作者简介

P. Taylor Goetz Apache Storm项目核心贡献者以及发布经理，自2011年10月Storm项目首次开源至今都参与其中，具有长期的Storm使用和开发经验。作为Storm用户社区中的活跃贡献者，Taylor领导了一系列开源项目，旨在使企业能够将Storm集成到不同的基础设施上。

Brian O'Neill 现就职于Health Market Science(HMS)公司，任首席技术官，重点进行数据管理和医疗领域数据分析。他已经担任技术主管超过15年，被公认为大数据领域的权威。作为系统架构师，他有着应对各种不同场景的经验，从初创公司到财富500强公司。他信奉开源精神，对多个项目做出了贡献。他领导的项目扩展了Cassandra数据库，并且将索引引擎、分布式处理框架、分析引擎集成到该数据库中。他荣获了2013年InfoWorld技术领导力大奖。

译者简介

董昭 腾讯安全平台部的应用运维安全工程师，负责腾讯Web业务的漏洞防护等相关工作，研究兴趣为网络安全、Linux后台开发、大数据等。

封面设计 · 王建敏

Storm是最流行的实时流计算框架之一，它提供了可容错分布式计算所要求的基本原语和保障机制，可满足大容量关键业务应用的需求。Storm不仅是一种集成技术，也是一种数据流和控制机制，已经成为很多大公司大数据处理平台的核心部分。

本书从简单的Storm topology示例出发，基于实际应用场景介绍Storm的基本功能，并详细讲解Trident和分布式状态等高级概念，以及与Druid和Titan的集成模式。通过阅读本书，读者将了解Storm和Trident的基本原理，并将这些原理和应用场景对应起来，解决实际问题。

通过阅读本书，你将学到：

- Storm的基本原理。
- 在伪集群模式和分布式集群模式下安装和配置Storm。
- 熟悉Trident和分布式状态的基本原理。
- 分布式系统数据流的设计模式。
- 与持久化存储机制的集成模式，如Titan。
- 利用YARN部署和运行Storm集群。
- 通过分布式存储获取持续可用性和容错机制。
- 了解集中日志机制和处理方法。
- 实现混合持久化和分布式事务处理。
- 通过点击流分析计算广告活动的效果。

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/大数据
ISBN 978-7-111-48438-7
9 787111 484387 >
定价：59.00元